

Treball de Fi de Màster

**Master in Automatic Control and Robotics**

# **Implementation of a Visual Servo Control in a Bi-Manual Collaborative Robot**

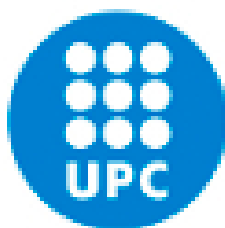
**Author:** José Agustín Aguilar Plazaola

**Director:** Jan Rosell Gratacòs

**Period:** 2014-2016

**Universitat Politècnica de Catalunya**

**Escola Tècnica Superior d'Enginyeria  
Industrial de Barcelona**





## ABSTRACT

This project presents the application of a visual servo control to an industrial human-like robot, both using a simulation environment and in a real platform. In a visual servo scheme, the control loop is closed by a vision sensor, usually a camera (or more than one, in the stereo approach). The camera acquires the image of a defined target and, a control algorithm calculates the relative pose of robot-target and then, continuously sends commands to the robot in order to position it as required. The pose calculation and controller algorithms have been written in C++. The work has been carried out through a sequence of stages that are presented in this document.

The first part goes through the basic theoretical ideas that support the design of the visual servo. It is composed of three main areas: computer vision, which deals mostly with the implementation of the vision sensor; robot kinematics, which allows define the equations that describe time evolution of the robot position, orientation, speed and joints values; and finally, the merge of both areas, the visual servo itself that makes up the control loop.

A next section explains how the different tools and frameworks have been used to implement the control loop. Some of these tools are manufacturer proprietary programs, others are open source. There is a detailed description of how the simulation environment is set, the content of each of the blocks in the control loop and a basic explanation of the manufacturers program.

The results show how the robot (simulated and real) converges to the relative set point pose and is also able to track changes in the position and orientation of the target.



## ACKNOWLEDGEMENTS

*To my son, Jose Eduardo, for constantly driving me to improve and show me the meaning of resilience.*

*To my wife, Yessica, for having made this immense effort and support me in everything I do.*

*To my mother, Gloria, who taught me the importance of education and knowledge.*

*To my parents in law, Jorge and Mercedes, for welcoming into their family like a son.*



# CONTENT

<b>ABSTRACT .....</b>	<b>3</b>
<b>ACKNOWLEDGMENTS .....</b>	<b>5</b>
<b>CONTENT .....</b>	<b>7</b>
<b>1. GLOSSARY .....</b>	<b>9</b>
<b>2. INTRODUCTION .....</b>	<b>11</b>
2.1. Industrial robotics .....	12
2.2. Collaborative robotics .....	12
2.3. Motivation of the project .....	13
<b>3. THEORY .....</b>	<b>15</b>
3.1. Computer Vision .....	15
3.1.1 Pinhole camera model .....	15
3.1.2 Image projection theory .....	16
3.1.3 Camera intrinsic matrix .....	17
3.1.4 Camera extrinsic matrix .....	19
3.1.5 Camera calibration .....	20
3.2. Robot Kinematics .....	21
3.2.1 YuMi forward kinematics .....	21
3.2.2 YuMi Jacobian matrix .....	23
3.3. Visual Servoing .....	24
3.3.1 Vision guided robotics .....	24
3.3.2 Interaction matrix (feature Jacobian) .....	26
3.3.3 Control law .....	28
3.3.4 Stability analysis .....	29
<b>4. IMPLEMENTATION .....</b>	<b>31</b>
4.1. Laboratory Setting .....	31
4.2. RobotStudio .....	32
4.3. RAPID .....	37
4.4. Gazebo .....	41
4.5. Target Patterns .....	47
4.6. ROS Nodes .....	48

4.6.1 Human joint limits and robot joint limits mapping .....	49
4.6.2 vp_simulation .....	50
4.6.3 converter_cv_ros .....	52
4.6.4 position_2.....	52
4.6.5 servo_simulation and servo_yumi.....	53
<b>5. RESULTS .....</b>	<b>57</b>
5.1. Simulation in Gazebo .....	57
5.2. Real robot.....	62
<b>6. ENVIRONMENTAL IMPACT .....</b>	<b>67</b>
<b>7. COST ANALYSIS .....</b>	<b>69</b>
<b>8. CONCLUSIONS AND FUTURE WORK.....</b>	<b>71</b>
<b>9. BIBLIOGRAPHY .....</b>	<b>73</b>



## GLOSSARY

DOF	Degrees of Freedom
ETSEIB	Escola Tècnica Superior d'Enginyeria Industrial de Barcelona
FK	Forward Kinematics
IK	Inverse Kinematics
IOC	Institute of Organization and Control
KDL	Kinematics Dynamics Library
OpenCV	Open Computer Vision Library
ROS	Robotic Operating System
SDF	Simulation Description Format
TCP	Tool Center Point
UPC	Universitat Politècnica de Catalunya
URDF	Universal Robot Description Format
ViSP	Visual Servoing Platform



## INTRODUCTION

Technology has played a leading role in the development of contemporary society; the world has seen a fast increase in technical outcomes during the past century that has even accelerated during its last decade and the first decade of the present one. Technical advances that have affected every aspect of life and every industry, from new energy efficient appliances at home to the increase in safety and decrease in cost in aeronautics, from tools and machines to introduce precision in agriculture to devices for more accurate diagnosis and faster recovery in healthcare, from faster networks and almost instant customer feedback in finance and service industries to reliable and cost effective automation in manufacturing.

An important part of those advances come from three main technical fields: information and computer science, electrical engineering (which, as a general field includes, energy, electronics and telecommunications studies) and mechanical engineering. And one of the most amazing products derived from this "technical revolution" and the conjunction of these three fields are, robots. A modern robot is a piece of equipment that brings together some of the leading edge knowledge in those areas.

The industrial robot that can be seen in today's more advances factories is part of a continuum, that stretches in time from centuries ago when many scientist and engineers, dreamed of mechanical servants; passes through more recent times, when even art and literature included them as one of its themes, as with the famous play written by Karel Capek, of which the word robot is derived from; and continues in present day, with the release of dozens new models of industrial robots a year, the research and development of robots for uses outside the factory, robots with shapes different than the arm-like of the

industrial one (parallel robots, quadruped, wheeled) and even robots for use beyond ground (drones, sea surface robots, underwater robots).

## 2.1 Industrial Robotics

The robots have found a perfect niche for its use in the structured environment of the manufacturing assembly line, nowadays they are an essential tool in many manufacturing facilities, being the car factory the most known application. Industrial robots are preprogrammed machines that rely on defined and expected behavior of the materials, tools and work flow of the factory, most of its movements are defined in the code and, in general, they cannot react to changes in that predictable behavior.

These industrial robots are usually installed, in a confined, fenced space, or at least, require the implementation of strict safety measures, such as restricted areas around them, emergency stop sensors, and several others, all this, to reduce the possibility of contact between the machine and the human workers, contact that most probably would cause injuries to the operators. There has been, recently, a rise in a new trend, the development of robots, that combining the precision and capacity of industrial robots, be able to work along human operators, moreover, assist them in its labors, a robot that interacts and collaborate with humans in an industrial setting.

## 2.2 Collaborative Robotics

That trend has been called **collaborative robotics**, and the machines produced for this aim, **cobots** (a contraction of those words). A cobot is a robot that, by design and construction, is inherently safe for working side by side with humans. There is no need of fences, or restrictions, the robot is able to physically interact with the worker without causing any harm.

These machines are embedded with sensors, that allow some maximum force and torque, so when in collision with an external object the robot immediately stops, this maximum levels, have been set at a lower level than the level known to harm a human body. The links and movable parts of these cobots are built with cushioned surface made of soft, yet, resistant materials, to further soften the force exerted when colliding with an external object, which, can probably be a human.

Cobots represent a forward step from industrial robots, however in order to become complete human coworkers and expand their use, a further step must be taken to widen the environments where they can operate and be ready to fully interact with the, in general, non predictability of humans.



**Fig. 2.1.** Some cobots currently available in the market, from left to right, Kuka LBR- iiwa, Universal Robots UR5 and Rethink Robotics Baxter.

## 2.3 Motivation and Objectives of the Project

As described above, one of the next steps along the line of this field, is to provide robots with the necessary capacities to perform tasks in non structured environments and operate safely side by side with humans workers. These capabilities are fundamental for robots to become full coworkers.

The main general motivation of the present project is then, to explore one of the control schemes that can help in the fulfillment of the above mention objective. This scheme is called visual servo control. Its central idea is to use visual feedback to close the control loop of the task being performed by the robot. It aligns with the procedure of adding sensors to the robot, so it can be aware of its environment and can react to unexpected (and then non pre-programmed) situations; the sensor, in the case of visual servo control, is a camera, or, as a generalization, any device that (with the aid of appropriate processing algorithms) can implement sensing to perceive color, depth, dimension, perspective, texture, etc, all without the need of physical contact.

A second, specific objective, is the implementation and test of the communication between the robot YuMi, which is an industrial, proprietary (and so, mostly closed) equipment and the open source framework, ROS (Robotic Operating System), once this communication has been achieved, the robot can be used for testing many other control schemes, and investigate in other areas of robotics, such as planning, which is a central research theme in the IOC Robotics Division.

The advancement in the use of Gazebo as a tool in the robotics laboratory can be cited as one secondary objective. The robot YuMi and the sensors required to test the control scheme are to be spawned in this virtual environment, so the algorithms can be tested before implementing them in the real robot; in this way, some of the many capabilities of the simulation tool are exploited and the possibility of error or damage when testing on the real platform is decreased. One of the most important aforementioned capability in Gazebo, is the option of simulating sensors in a simple manner, setting its parameters and even include some of its disturbances such as noise. Another worth to mention feature, is that, it provides with procedures to tune or even create custom defined low level joint controllers.

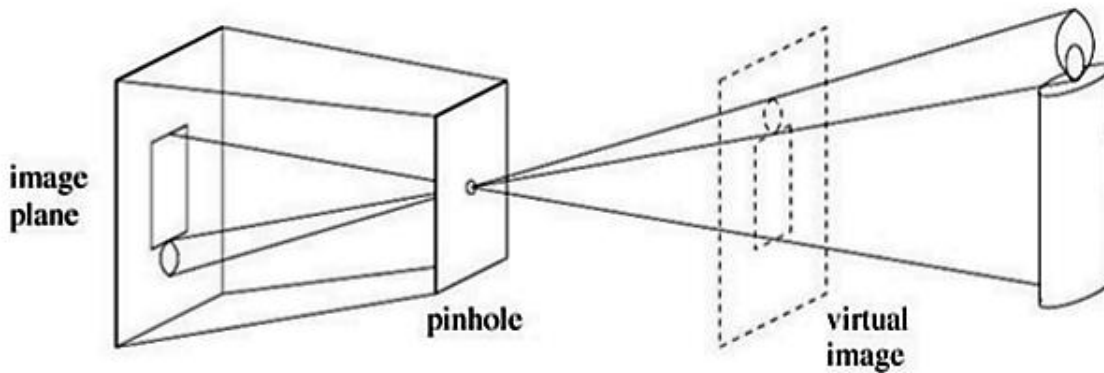
# THEORY

## 3.1 Computer Vision

### 3.1.1 The pinhole camera model.

The camera is the sensor used in this project to close the high level control loop that will direct the trajectories of the robot. A modern digital camera is composed of several parts and is of a large complexity, however, in general, any camera (including analogical) can be represented by two fundamental sections that perform the most basic functions expected from an imaging device: first, receive the light rays from the exterior and direct those rays in a certain direction, and second, reproduce (and record) those rays as an image on some surface and in a form that can be manipulated, this object would be a photosensitive film in an analogical camera or a matrix of small light sensors in a digital one.

A simple way of modeling the actions performed by those two parts is through the well known **pinhole camera model**. It is based in the primitive artifacts used to produced images centuries ago, a simple closed box with a small hole in the center of one of the sides. The light rays enter the box through the pinhole and project the image in the internal surface of the opposite side of the cube, the image plane. One of the disadvantages of this model, is the fact that the image plane reproduce an inverted version of the target object, so it is usual to define a virtual image plane located in front of the camera at the same distance from the pinhole as the image plane, this virtual plane contains a non inverted version of the target (Figure 3.1).



**Fig. 3.1.** The pinhole camera model [6].

This imaging model is called **perspective projection**; from this simple representation can be derived equations that relate many important variables in the relationship between the camera (with its light concentration and image formation parts) and the target object.

### 3.1.2 Image projection theory.

It seems clear from the previous description that the analysis should be focused in three elements, the pinhole, the image plane and the target object.

Figure 3.2 shows the schematic depiction; the pinhole is called the principal point, a camera coordinate system ( $Oijk$ ) is defined with its origin in this principal point, then, a target object  $T$  has a specific point  $P$ , which is projected in the image plane, as a point  $p$  by a single light ray passing through the principal point, the distance between the image plan and the pinhole, is the focal distance  $f$ ; the horizontal line that goes from the object, through the focal point and reaches the image plane is called the optical axis. The Euclidean space coordinates of the point  $P$  with respect to the camera coordinate system are  $X, Y, Z$ , and the coordinates of the point  $p$  in the image space are  $x, y, z$ . The optical axis lies in the same line as the  $z$  axis.

Since the points  $P$ ,  $O$  and  $p$  are collinear, the relation  $Op = \lambda OP$  holds for any  $\lambda$ , then:



$$x = \lambda X$$

$$y = \lambda Y$$

$$f = \lambda Z$$

By solving for  $\lambda$ :

$$\lambda = \frac{x}{X} = \frac{y}{Y} = \frac{f}{Z}$$

Then:

$$x = f \frac{X}{Z}; \quad y = f \frac{Y}{Z} \quad (3.1)$$

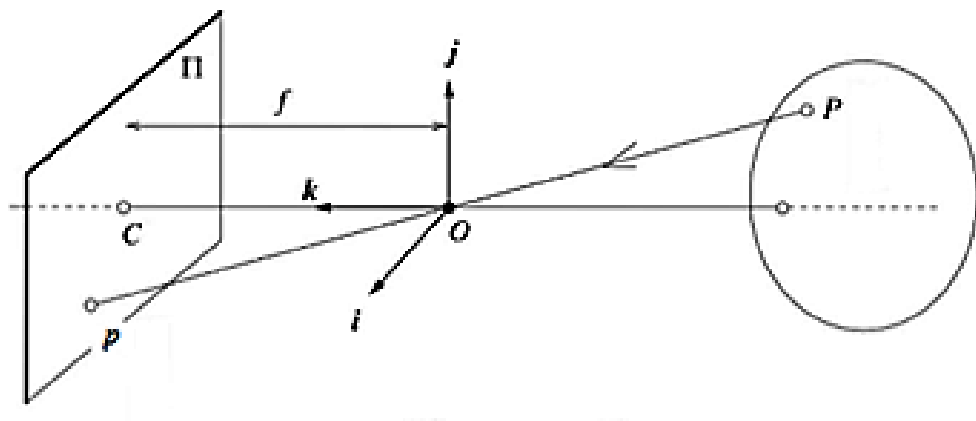
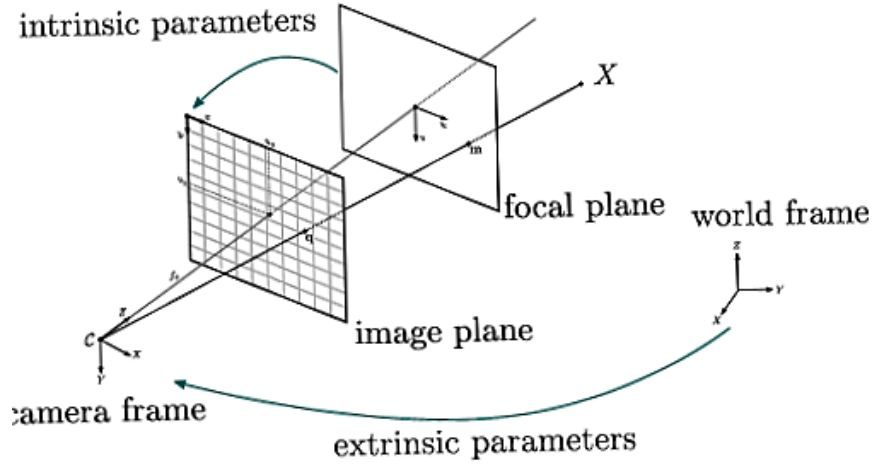


Fig. 3.2. Perspective projection process [6].

### 3.1.3 Camera intrinsic matrix

Starting from the perspective projection model, the objective is then to find all the parameters that do the mapping of a target object, from its coordinate description in the to the coordinates of its projection in the image space. These mapping parameters can be divided in two parts, first, the transformation that relates the world coordinates of the target to the camera system coordinates, and second, the parameters that transform those camera

system coordinates to image space coordinates. They are called, extrinsic and intrinsic parameters respectively.



**Fig. 3.3.** Virtual image plane and pixel coordinates [6].

A point in space, projected in the image plane, is specified by coordinates  $[u, v]$  measured in pixels. To avoid the problem of image inversion in the analysis of the projection model, a virtual plane is assumed to exist in front of the principal point (pinhole) at the same focal distance than the real image plane. From equation (3.1), the relation that maps from Euclidean to image space can be obtained as detailed in [6]:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (3.2)$$

Where in general:

$[X, Y, Z]$  are the meter coordinates of the point in the camera frame.

$[u, v]$  are the pixel coordinates of the projection point in the image plane.

$[c_x, c_y]$  are the pixel coordinates of the principal point.

$(f_x, f_y)$  are the focal distance in the  $x$  and  $y$  direction, expressed in pixels.

$s$  is a scale factor.

The mapping done by equation (3.2) returns the pixel coordinates of the projection point  $p$  in the image space, from the coordinates of the target point  $P$  that are expressed in the camera system. The focal length and principal point coordinates are the intrinsic parameters of the camera model.

### 3.1.4 Camera extrinsic matrix

The point  $P$  that has been mentioned though the analysis, can be thought as a specific point in space of a body or in general, in a scene being recorded by the camera. This point reflects one ray of light that impact the image plane in the specific point  $p$ . In general all the objects in a scene are expressed in a fixed world coordinate system with an appropriate pose (position and orientation). In order to express the  $[X_w, Y_w, Z_w]$  world coordinates of  $P$  in the coordinates of the camera frame, a homogenous transformation is required. The extrinsic parameters compose the rotation matrix and translation vector that perform this transformation.

As expressed in [6]:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = [R|t] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \quad (3.3)$$

Where in general:

$[X, Y, Z]$  are the meter coordinates of the point in the camera frame.

$[X_w, Y_w, Z_w]$  are the meter coordinates of the point in the world frame.

$R$  is a rotation matrix that brings point from the world frame to the camera frame.

$t$  is a translation vector that brings points from the world frame to the camera frame.

### 3.1.5 Camera calibration

The intrinsic and extrinsic parameters of a camera, whose relation equations has been stated, allow mapping the three dimensional Euclidean space into the two dimensional image space. This mapping is of vital importance in visual servo control because the camera is the sensor from which the error between a desired target (or camera) pose and a current pose is calculated. That is, if all the camera parameters are known, and, the  $[u, v]$  image space coordinates of a target point are obtained, then, using all the shown equations, its  $[X_w, Y_w, Z_w]$  world space coordinates can be obtained at any instant.

Camera calibration is the process of finding those parameters; together they form what is called the fundamental matrix of the camera,  $\mathcal{M}$ . Following the development in [6]:

$$\mathcal{M} = \mathcal{K}[R|t] \quad (3.4)$$

Where  $\mathcal{K}$ , represents the matrix of intrinsic parameters.

A calibration algorithm finds the image coordinates of  $n$  fiducial points,  $P_i$  ( $i = 1, 2, \dots, n$ ) with known Euclidean coordinates  $\mathbf{P}_i$ , assuming no error measurements, the camera parameters can be calculated by solving equation (3.5) for  $\xi$ :

$$u_i = \frac{m_1(\xi) \cdot \mathbf{P}_i}{m_3(\xi) \cdot \mathbf{P}_i}, \quad v_i = \frac{m_2(\xi) \cdot \mathbf{P}_i}{m_3(\xi) \cdot \mathbf{P}_i} \quad (3.5)$$

Where,  $m_i(\xi)$  represents the  $i^{\text{th}}$  row of the fundamental matrix and  $\xi$  is the vector of extrinsic and extrinsic parameters.

Most calibration algorithms treat this as an optimization problem, where the position error between the found images coordinated of the fiducial marks and its theoretical position is minimized with respect to the extrinsic and intrinsic parameters. The fiducial points come from some defined pattern, where specific features can be easily found. These features can be for example, the corner points in a chessboard or the centers of regularly distributed black blobs in a white background.

An existing C++ code in OpenCV (Open Computer Vision Library) and a chessboard pattern were used in this project to find the camera parameters.

## **3.2 Robot Kinematics**

### **3.2.1 YuMi forward kinematics**

A manipulator robot is a kinematic chain of rigid segments and articulations named links and joints respectively, in a non theoretically strict manner, the number of joints is said to be the number of Degrees of Freedom (DOF) of the robot. By convention, at the end of the final link is attached the coordinate system that specifies the position and orientation of the manipulator. In the area of industrial robotics, the end of that final link is called the Tool Center Point (TCP) of the robot; if the robot is carrying a tool, the TCP would be then located in the useful point of the tool.

Each of the six variables that define the pose of the TCP (three linear coordinates for position and three angles for orientation) result from a parameterized equation whose parameters correspond to the value of the joints (the joint value units are radians or degree for revolute joint and meters for linear joints) at a given instant. These equations form the Forward Kinematics (FK) description of the robot.

$$\begin{bmatrix} \mathbf{P}(t) \\ \mathbf{W}(t) \end{bmatrix} = \mathcal{F}(q_1(t), q_2(t), q_3(t), q_4(t), q_5(t), q_6(t), q_7(t)) \quad (3.6)$$

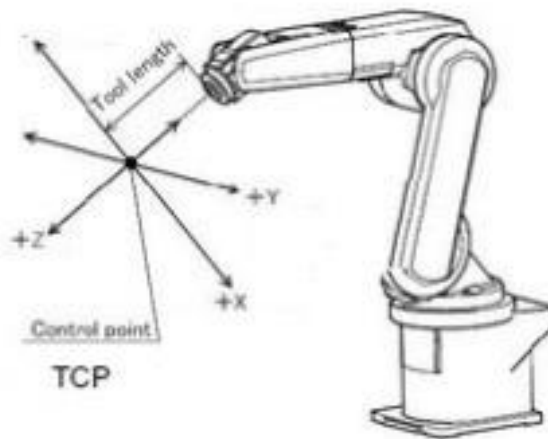
Where:

$\mathbf{P}(t)$  is a 3x1 time variant vector that defines the position of the TCP.

$\mathbf{W}(t)$  is a 3x1 time variant vector that defines the orientation of the TCP.

$\mathcal{F}$  is a vectorial function.

$q_i(t)$  represent each joint of the robot.



**Fig. 3.4.** TCP coordinate system [3].

The robot used in this project, commercially called YuMi, is a dual arm manipulator manufactured by the robotic company ABB (Figure 3.5), it was released to market during 2015 after several years of research, it is then, a piece of equipment at the edge of technological development. It is directed toward the rising field of collaborative robotics, and designed to work in a safe manner along human workers, thus it has human range movement speed and it stops the movement once it feels the slightest collision.

In the sense of more accurately imitate the movements of its future work companions, YuMi, has seven DOF distributed as an human arm does, so it can place and orientate its TCP in the full range of the 3D space and also count on an extra DOF to avoid a possible collisions in its trajectory to the target pose.



**Fig. 3.5.** The robot YuMi, by ABB.

### 3.2.2 YuMi Jacobian matrix

The FK equations can be differentiated with respect time and then the relation between 3D space velocities and joint velocities is found. This is a convenient way of expressing and working on the manipulator kinematics because it allows grouping variables and end with a nice expression of just two vectors related by a matrix, this matrix is called the Jacobian of the robot.

$$\begin{bmatrix} \dot{\mathbf{P}}(t) \\ \dot{\mathbf{W}}(t) \end{bmatrix} = \mathbf{J}(\mathbf{q}_1(t), \mathbf{q}_2(t), \mathbf{q}_3(t), \mathbf{q}_4(t), \mathbf{q}_5(t), \mathbf{q}_6(t), \mathbf{q}_7(t)) \dot{\mathbf{Q}}(t) \quad (3.7)$$

Where:

$J$  is a 6x7 Jacobian matrix that depends on each joint value.

$\dot{Q}(t)$  is a vector of joint angular velocities.

The Jacobian is of capital use in the project's control strategy, because, as it will be seen, the control signal delivered by the servoing algorithm is a  $\mathbb{R}^3$  space velocity command and thus, finding the inverse of the Jacobian in equation allows mapping that command to the joint space of the robot. The solution of the Jacobian is done through an existing function in the used programming language library.

## 3.3 Visual Servoing

### 3.3.1 Vision guided robotics

Computer vision tools and devices have been introduced, relatively not so long ago, as a new tool to provide some sensing capabilities to industrial robots. A typical application of such tools is the installation of a camera, above a conveyor belt, placed in a fixed position upstream of a robotic cell that contains a delta robot. The camera points down and looks at a particular area of the conveyor belt where some targets (usually light small items, for example cookies in a packaging application) that must be picked from the transport line and placed in a specific point by the robot, pass. The transformation between the camera frame and the robot frame is fixed and known, so it is the speed of the belt. The vision sensor calculates the position of the target with respect to its frame, sends this information to the robot controller and, by the known transformations and speed, the controller calculates the goal pose of the robot TCP and the instant when it must be reached, to pick the item.

This assembly is called **vision guided robotics**. It is clear that the configuration does not provide a closed loop control, that is, the sensor sends a pose and the controller execute the



movement, but if for some reason, the cookie changes its position once it has passed the camera and before it gets to the cell, the robot will still move to the commanded position, even though there is no cookie there. One of the main objectives for developing the visual servoing framework is to cope with these unexpected changes that usually occur in non-structured environments (and in many different applications than a cookie factory).

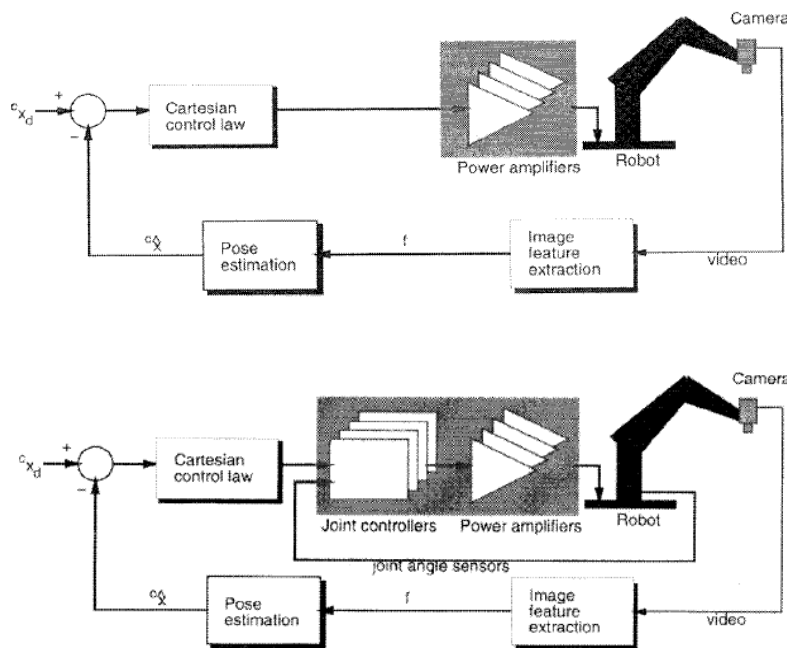
A visual servo scheme closes the loop formed by the vision sensor, the control and the robot. In the industrial setting, the controller would receive, in “real time”, the pose (either in Euclidean space or in a space defined by the image features, as will be explained later) of the target, and will compare this pose with a reference, set by the operator, send continuous (continuous in the sense of a high sampling rate, instead of a single goal pose) motion commands to the robot and then correct its TCP pose at every instant.

In the literature [4], [7], there is further distinction of the visual servoing strategy in three types. The “**pure**” **visual servo**, in which the control block directly commands the actuators of each robot joint, so there is not a low level control and only the outer high level loop exists, the camera is the only sensor present in the scheme. Then there is a “**middle**” **visual servo**; in this case, the commands from the visual control block are received by low level controllers that handle the exact positioning of the joint, so there are low level control loops for each actuator, formed by its own control block (a PID is usually implemented as the controller), the joint mechanism, and an angular sensor (an encoder for example). And finally, a strategy called “**look-then-move**” that lays halfway from the “middle” type servo and the vision guided, it has a low command rate but still sends vision produced trajectory points between initial and goal pose, and checks through the camera, whether the target has been reached or not at every sampling point.

This project implements a “middle” type visual servoing strategy in which the higher level control loop sends commands to the inner low level loop that controls every joint. In the case

of the real robot, this low level control is done by the ABB industrial controller, for the virtual one, it is performed by special features added in the simulation environment, as will be detailed later.

The configuration of the arrangement robot-sensor-target can also be interchanged, thus, producing another classification for the control strategy. The camera can be hold by the robot in a fixed pose with respect to its TCP, then the robot moves the camera following the movements of the target, this is called an **eye-in-hand** scheme, but also, the camera can be placed in a fixed pose with respect to the world frame (and so the robot base frame) and the target being hold by the robot, in this case the robot moves the target trying to position it in a desired way with respect to the camera, this is the **eye-to-hand** configuration.



**Fig. 3.6.** Schematic diagrams of “pure” position based visual servoing and “middle” type position based visual servoing [7].

### 3.3.2 Interaction matrix (the feature Jacobian).

The pose of the target (and also the goal pose) can then be described in two different

spaces, either in an Euclidean space, for which then the pose is characterized by a position vector and any of the formalisms for describing rotation, either matrix, angle-vector, Euler angles or quaternion, this is the **position-based visual servoing** (PBVS), or, in an image feature space, in this case, some chosen features of the target image are tracked and an array of goal features is defined, the TCP will move towards converging the instantaneous features to that goal array, **image-based visual servoing** (IBVS).

The following development is detailed in [1]. The visual servo algorithm seeks to minimize the error,  $e(t)$ , expressed in equation (3.7) as the difference between the current value of some feature vector  $s$  and a desired feature vector  $s^*$ . The first one is dependent of two main variables: some vector of image measurements,  $m(t)$ , which are defined in an Euclidean space or an image space, mentioned in before; and, a vector of constant parameters of the system,  $a$ .

$$e(t) = s(m(t), a) - s^* \quad (3.7)$$

From that equation can be derived the control law for the visual servo as a velocity controller. Since the desired feature vector is constant, the rate of change in the error is:

$$\dot{e}(t) = \dot{s}(m(t), a) \quad (3.8)$$

Also, the image moves (and so the rate of change of the features vector) in dependence with the velocity of the TCP (and assuming from now an eye-in-hand scheme), meaning that the rate of change of the features is related with the camera velocity, that is:

$$\dot{s}(t) = L_s \mathcal{V}_c \quad (3.9)$$

Where,  $\mathcal{V}_c$ , is a 6x1 vector that denotes the velocity of the camera and is formed of its linear  $v_c$  and angular  $\omega_c$  velocities, and  $L_s$  is the matrix that relates that velocity with the rate of

change of the image features. This matrix is often called interaction matrix or feature Jacobian.

By merging (3.8) and (3.9) , the relation between error time change and camera velocity is found:

$$\dot{e}(t) = L_e \mathcal{V}_c \quad (3.10)$$

Where  $L_e = L_s$ . This matrix is also, in general, time dependant.

### 3.3.3 Control law

If the camera velocity is considered as the control input of the system, by seeking a exponential decrease of the error in equation (3.10), the obtained control law is:

$$\mathcal{V}_c(t) = -\lambda L_e^+ e(t) \quad (3.11)$$

Where  $L_e^+$  is the Moore-Penrose pseudo inverse. Since interaction matrix is time dependant, so it is its pseudo inverse, and then this matrix must be estimated at every instant (just as it is the kinematic Jacobian matrix) for producing the control signal.

In the case of the PBVS approach the feature vector  $s(t)$  can be described by the instantaneous vector of translation of the camera position  $t(t)$  and the instantaneous rotation of the image,  $\theta u(t)$  (in angle-axis parametrization). So:

$$\begin{aligned} s(t) &= (t_o^c(t), \theta u(t)) \\ s^* &= (t_o^{c*}(t), 0) \end{aligned} \quad (3.12)$$

$$e = (t_o^c(t) - t_o^{c*}(t), \theta u(t))$$

The interaction matrix is found to be:

$$L_e = \begin{bmatrix} -I_3 & [t_o^c]_x \\ \mathbf{0} & L_{\theta u} \end{bmatrix} \quad (3.13)$$

Where  $I_3$  is the identity 3 x3 matrix and  $L_{\theta u}$  is given by [6] as:

$$L_{\theta u} = I_3 - \frac{\theta}{2} [\mathbf{u}]_x + \left(1 - \frac{\text{sinc}\theta}{\text{sinc}^2 \frac{\theta}{2}}\right) [\mathbf{u}]_x^2 \quad (3.14)$$

Where  $\text{sinc}x$  is defined as  $x \text{sinc}x = \sin x$  and  $\text{sinc}0 = 1$ .

Finally this gives the control law:

$$\begin{aligned} \mathbf{v}_c &= -\lambda R^T \mathbf{t}_c^*(t) \\ \boldsymbol{\omega}_c &= -\lambda \theta \mathbf{u}(t) \end{aligned} \quad (3.15)$$

### 3.3.4 Stability analysis

The stability of the system can be assessed using Lyapunov analysis. By taking the squared error norm as a candidate Lyapunov function:

$$\mathcal{L} = \frac{1}{2} \|\mathbf{e}(t)\|^2 \quad (3.16)$$

The development yields:

$$\dot{\mathcal{L}} = \mathbf{e}^T(t) \dot{\mathbf{e}}(t) \quad (3.17)$$

And from (3.10) and (3.11):

$$\dot{\mathcal{L}} = -\lambda \mathbf{e}^T(t) L_e L_e^+ \mathbf{e}(t) \quad (3.18)$$

So, in order to assure global asymptotic stability, a sufficient condition is that:

$$L_e L_e^+ > 0 \quad (3.19)$$

Because  $\mathbf{L}_{\theta u}$  given in (3.14) is nonsingular when  $\theta \neq 2k\pi$ , the global asymptotic stability of the system is obtained, since

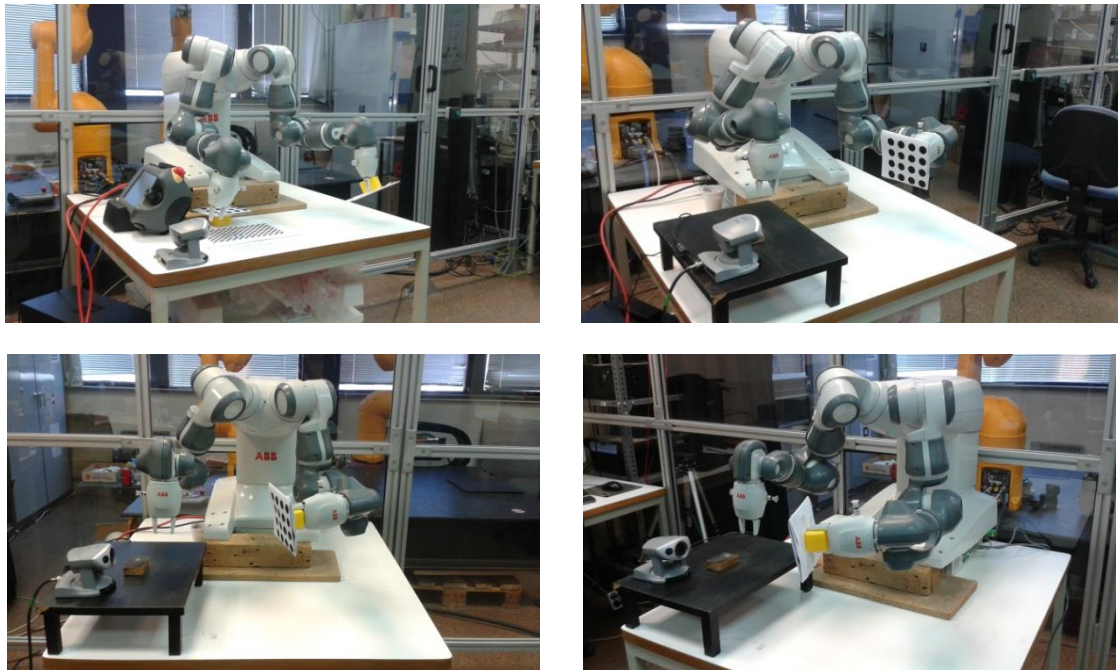
$$\mathbf{L}_e \mathbf{L}_e^+ = \mathbf{I}_6 \quad (3.20)$$

# IMPLEMENTATION

## 4.1 Laboratory Setting

The real setting for testing the control schemes and algorithms of the project is located in the IOC Robotics Laboratory. It is composed of three PCs, a bi-manual robotic manipulator (YuMi), its industrial controller, a CANON camera and a series of marker patterns as targets. One of the PCs, named WIN PC from now, has Windows as its operating system and runs RobotStudio and Rapid, another PC, operates on Debian system, has ROS installed and runs all the nodes with the sensing and control algorithms, this will be ROS PC, the third PC has a special hub card that allows connect with the special camera, CAM PC.

The robot is installed above a working table where also the camera is located, and since the scheme is an eye-to-hand, the robot holds the target in one of its grippers (Figure 4.1).



**Fig. 4.1.** Laboratory setting.

## 4.2 RobotStudio

RobotStudio is the simulation environment and programming platform for ABB robots. The manufacturer provides this tool that allows off-line testing of routines, load and download those routine to and from the controller, creation of virtual robotics stations with all the equipment and settings that can be found in the real factory, and also, a graphical user interface for communicating with the real robot and check, on-line, the execution of the program loaded in the controller.

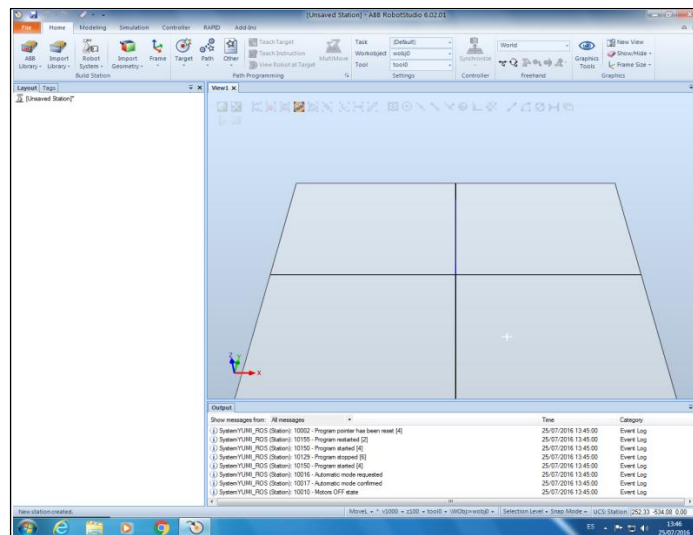


Fig. 4.2. RobotStudio environment displaying an empty station

For testing the routines that allow communication and execute the motion commands in the real robot, a station, with the manufacturer virtual version of YuMi, has been created. This is done by first loading the mechanism and the grippers using the *ABB Library* and *Equipment Library* option of the *Home* tab that appears in the menu ribbon located in the upper part of the screen (Figure 4.3).

With the mechanism installed in the station, a controller must be chosen to move the robot. The *Robot System* option is opened; the interface can automatically find a controller that match the mechanism of the station



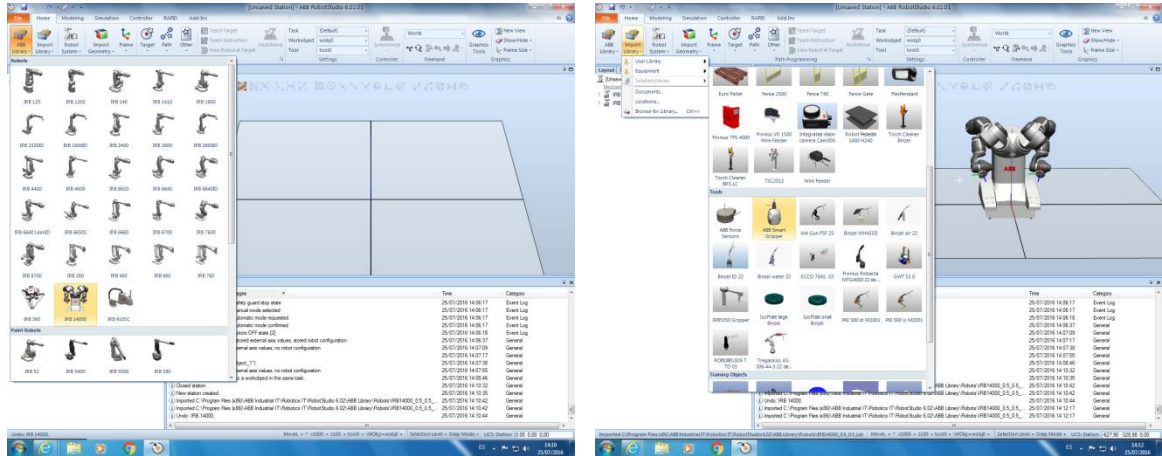


Fig. 4.3. Setting the mechanism in the simulated station

Once the robot and the controller have been created, some important options must be configured in the controller to allow communication and tasks execution. This options appear when setting the controller, the most important for the project are: *Multitasking*, which allows running several tasks in parallel having some of them as background tasks, and *PC Interface*, that enable the capacity of communicating with another PC through the network by establishing socket connections (Figure 4.4).

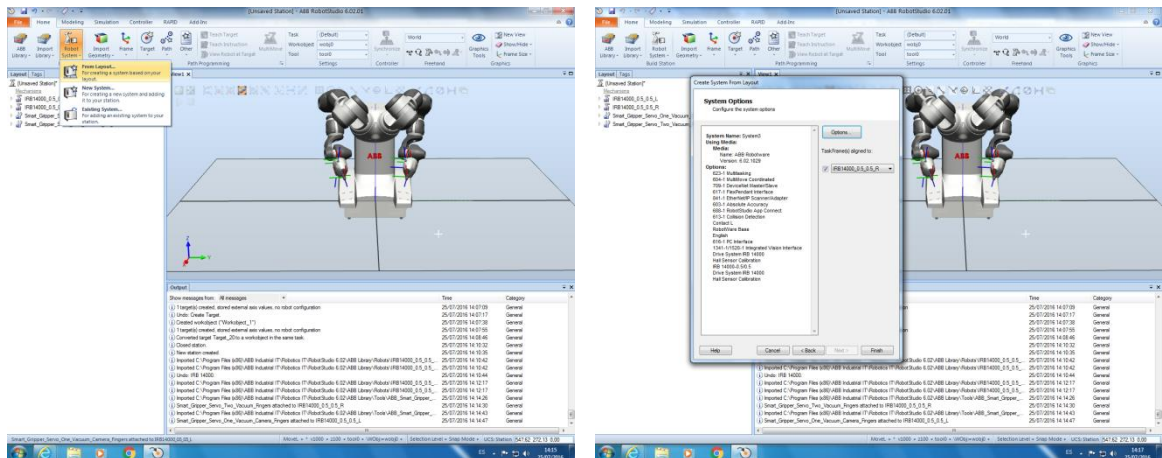
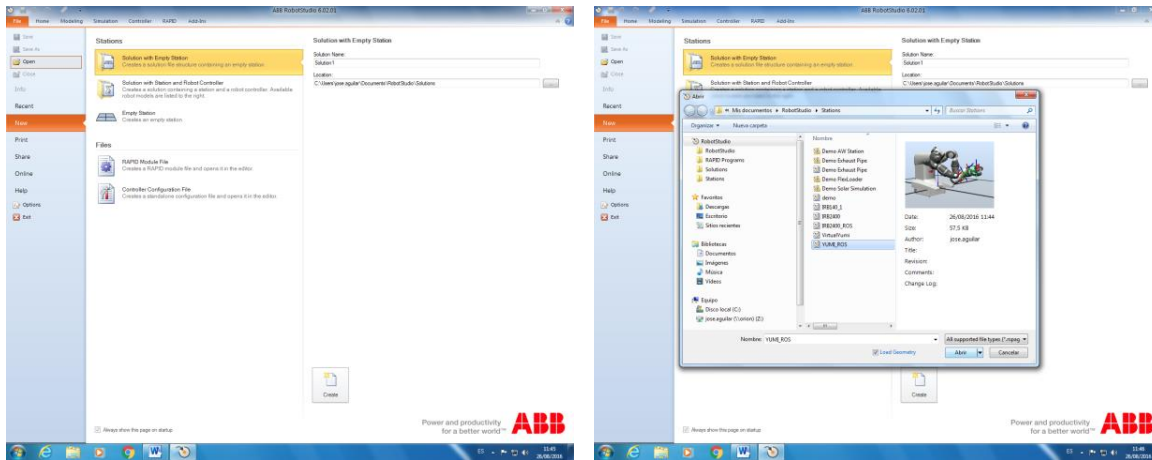


Fig. 4.4. Setting the mechanism in the simulated station

For the project, a station called **YUMI\_ROS** has been created in the environment. When opening the application, the *Open* option is chosen and the list of already created stations will

appear (Figure 4.5).



**Fig. 4.5.** Opening the station

Once the station has been loaded in the environment, the initialization status of the controller is shown in the bottom right corner; it will be ready to operate when the yellow rectangle becomes green. If, after some seconds it does not turn green, a message will appear in the log list indicating an error in the start up. This can be fixed by opening the *Controller* tab in the menu ribbon, and then open the *Control Panel*. A virtual panel appears in the right side, displaying an On-Off handle, an emergency button and a key to specify whether the robot will operate in manual or automatic mode. By clicking it twice, it will turn to the *Off* position and then to the *On position* again, the mode key changes to *Auto*, and the bottom right rectangle that shows the controller status becomes green. In the left side of the window, a hierarchical menu is seen, showing several contents, the most important for the work that will be done with the robot is the *Rapid* label, there, the program that is loaded in the simulated station can be opened and edited (Figure 4.6).

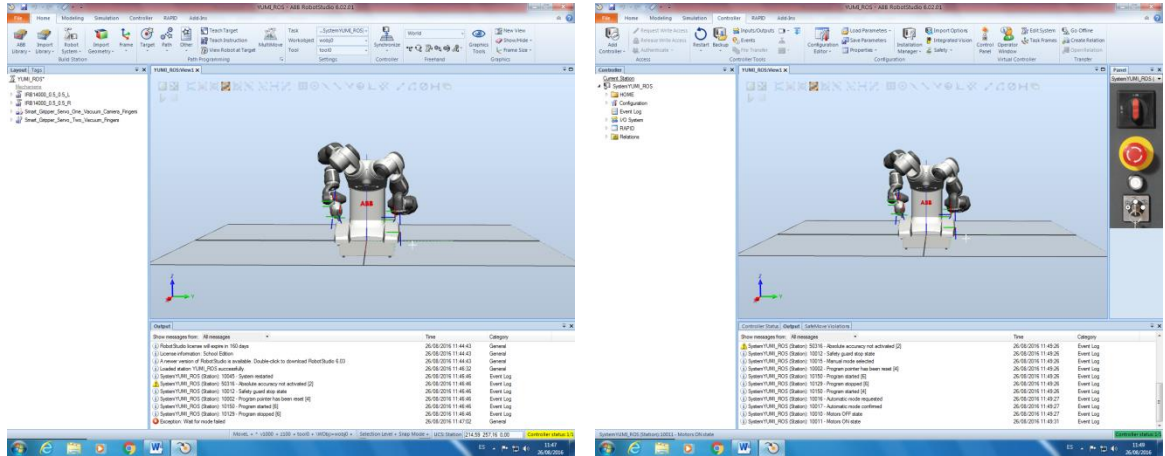


Fig. 4.6. Starting the controller.

With this, the simulated station is ready to operate. For connecting RobotStudio with the real robot, being in the *Controller* tab, the *Add Controller* option is to be clicked. The network will be scanned to find the industrial controller, a window appears showing the system name (the code name in the ABB nomenclature of the model), the controller name, its IP address and the version of the operating system running in it, clicking OK will establish the connection (Figure 4.7).

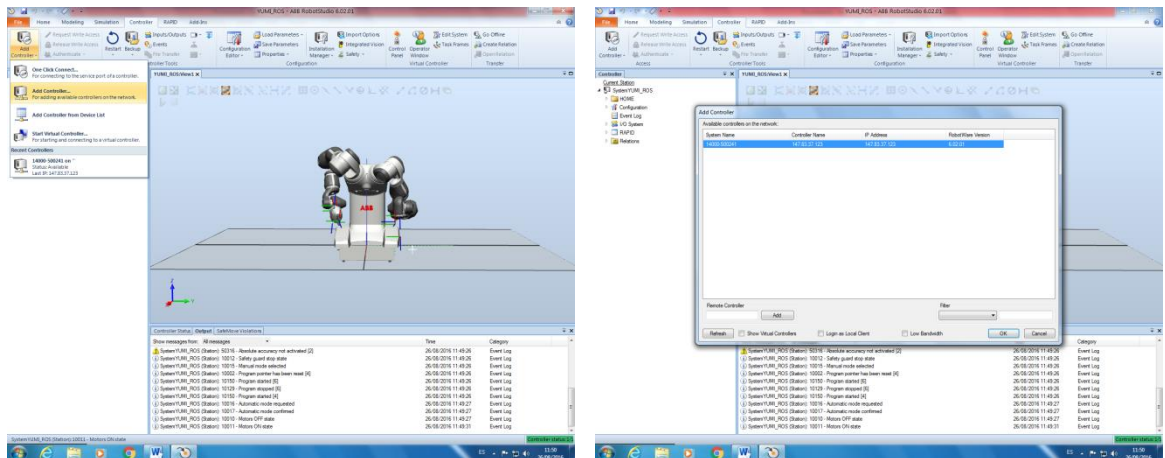
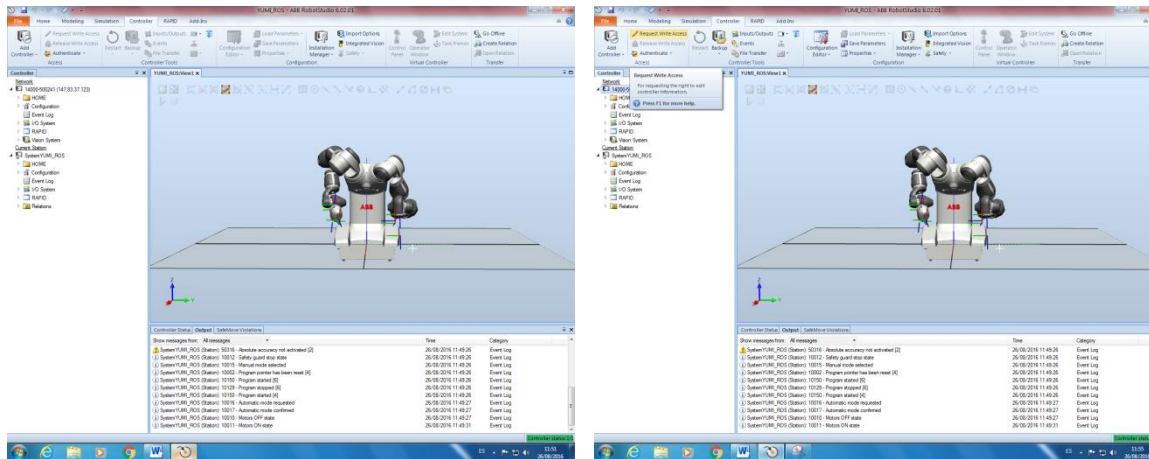


Fig. 4.7. Connecting to the real industrial controller.

Once the connection is set properly, the hierarchical menu of the real robot appears in the left side (either above or below than the one of the simulated station). The graphical interface

is connected to the robot, then, by opening the *Rapid* menu the execution of the program can be followed. For debugging, editing and gaining control over the run of the code, further steps must be followed. The automatic mode must be specified in the Teach Pendant of the robot, then, in the *Controller* tab, the option *Request Write Access* is selected, with this, the program can be edited and its execution controlled from RobotStudio (Figure 4.8).



**Fig. 4.8.** Allowing access from the graphical interface

For viewing the code, control its execution and open several options for debugging it, the *Rapid* label in the hierarchical menu is opened, it will show lower parts in the hierarchy of the program (this hierarchy will be explained in a later section), and also open the menu ribbon of the *Rapid* tab in the upper part of the screen.

One of the control and debugging options is the *Program Pointer*, the position where the operator wants to start execution, for example, putting the pointer in the main process in all the tasks (which is the default option when the code starts) or by locating the pointer in a specific line of the program and clicking the *Play* option (Figure 4.9).

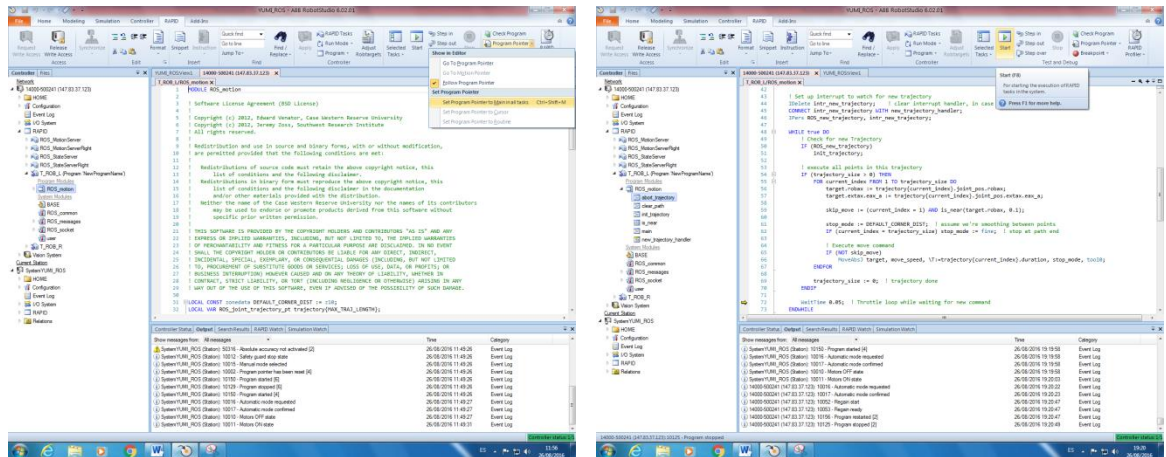


Fig. 4.9. Viewing the code, setting the pointer and run.

## 4.3 RAPID

RAPID is the proprietary high-level language of ABB for programming its robots. Figure 4.10 shows the structure of the program used in the project.

```

BACKGROUND TASKS

ROS_MotionServer
Module ROS_motionServer
main()
receive_joint_trajectory_processes()

Task_ROB_L
Module ROS_motion
main()
execution_processes()

Task_ROB_R
Module ROS_motion_right
main()
execution_processes()

Module ROS_stateServer
main()
send_joint_trajectory_states_processes()

ROS_MotionServerRight
Module ROS_motionServerRight
main()
receive_joint_trajectory_processes()

Module ROS_stateServerRight
main()
send_joint_trajectory_states_processes()

```

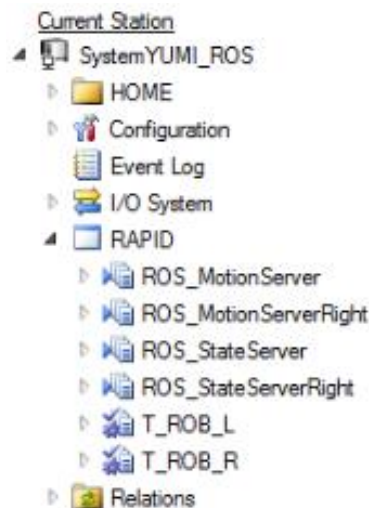
Fig. 4.10. Front and background tasks of the RAPID program.

A program in Rapid can be broke down in several layers: Tasks, Modules, Processes and Functions. Tasks are at the higher level division; generally a robot program consists of a few tasks to be performed by the mechanism. When assembling the code, these tasks can be defined as Static or Semi-Static, in the case of the first ones, the task is visible to all the



RobotStudio editing options and can be accessed in the robot Teach Pendant for making changes in the code, the second type of tasks run in “background execution”, they are not visible to several editing menu options in the user interface and cannot be edited in the Teach Pendant. These Semi-Static tasks do the function of running code that in parallel to the actual motion routines that must be done by the robot; for example, handling external I/O signals that may affect the front code that runs in the Static tasks.

For the purpose of the project, that capability is fundamental since YuMi is to be controlled from an external source, which is ROS. While the motion execution tasks run in the front of the robot controller, there are two main Semi-Static tasks handling the socket communication with the ROS PC and constantly sending and receiving messages to and from the ROS Nodes, through the network. These are the ROS\_MotionServer and ROS\_StateServer tasks for the left arm mechanism, and ROS\_MotionServerRight and ROS\_StateServerRight for the right arm mechanism.



**Fig. 4.11.** Static and Semi-Static tasks running in YuMi.

Figure 4.11 shows how RobotStudio marks which of the tasks are Static (thus running in the front) and also motion tasks, by drawing a check sign and a gear icon on them. Those are

T\_ROB\_L and T\_ROB\_R, they have the actual motion routines to be executed by the robot. These routines are contained in the modules ROS\_motion and ROS\_motion\_right for the left and right arm respectively.

It is worthwhile to see the content of some parts of those modules to understand, at least at basic level, how the program is built. Each module is specified with the label *MODULE* followed by its name. At the beginning of the module the main constants and variables are defined. In the case of the ROS\_motion module, some of those are, the *zonedata*, which indicates how far from a specific target point the robot will pass; *trajectory\_size* which will define the number of trajectory points received for execution; *current\_index*, a pointer to the current trajectory point being executed, *move\_speed*, the motion speed of the robot; *target*, a *jointtarget* type variable, which is the actual set point to which the robot joints must move; and others. After the constants it also can be seen the definition of the followed part as the main execution process by the label *PROC main()* (Figure 4.12).

```

MODULE ROS_motion

LOCAL CONST zonedata DEFAULT_CORNER_DIST:=z10;
LOCAL VAR ROS_joint_trajectory_pt trajectory{MAX_TRAJ_LENGTH};
LOCAL VAR numtrajectory_size:=0;
LOCAL VAR intrnum intr_new_trajectory;

PROC main()
VAR num current_index;
VAR jointtarget target;
VAR speeddata move_speed:=v500; !default speed
VAR zonedata stop_mode;
VAR bool skip_move;

```

**Fig. 4.12.** Variables definition in module ROS\_motion

The main motion execution loop is a permanent *while* command that checks whether a new trajectory point has been received, runs through all the elements in the trajectory vector and, comparing those points with a defined tolerance decides if the robot must move, or if it is already close enough to the target point, those points are joint position targets received through socket connection by other modules.

```

WHILE true DO
! Check for new Trajectory
IF (ROS_new_trajectory)
init_trajectory;

! execute all points in this trajectory
IF (trajectory_size > 0) THEN
FOR current_index FROM 1 TO trajectory_size DO
target.robax := trajectory{current_index}.joint_pos.robax;
target.extax.eax_a := trajectory{current_index}.joint_pos.extax.eax_a;

skip_move := (current_index = 1) AND is_near(target.robax, 0.1);

stop_mode := DEFAULT_CORNER_DIST; ! assume we're smoothing between points
IF (current_index = trajectory_size) stop_mode := fine; ! stop at path end

! Execute move command
IF (NOT skip_move)
MoveAbsJ target, move_speed, \T:=trajectory{current_index}.duration, stop_mode, tool0;
ENDFOR

trajectory_size := 0; ! trajectory done
ENDIF

WaitTime 0.05; ! Throttle loop while waiting for new command
ENDWHILE

```

**Fig. 4.13.** Motion execution loop.

The ROS\_motionServer module is in charge of receiving the messages, and constructing the vector with the trajectory points by calling some local process defined in it. Its structure is similar to the previous module, it creates the constants and variables that will be used such as *server\_socket* and *client\_socket* which are of type *socketdevice*.

```

MODULE ROS_motionServer

LOCAL CONST numserver_port := 11000;

LOCAL VAR socketdevserver_socket;
LOCAL VAR socketdevclient_socket;
LOCAL VAR ROS_joint_trajectory_pt trajectory{MAX_TRAJ_LENGTH};
LOCAL VAR numtrajectory_size;

PROC main()
VAR ROS_msg_joint_traj_pt message;

TPWrite "MotionServer: Waiting for connection.";
ROS_init_socketserver_socket, server_port;
ROS_wait_for_clientserver_socket, client_socket;

WHILE (true) DO
! Recieve Joint Trajectory Pt Message
ROS_receive_msg_joint_traj_ptclient_socket, message;
trajectory_pt_callback message;
ENDWHILE

```

**Fig. 4.14.** ROS\_motionServer module.



```

MODULE ROS_stateServer

LOCALCONST numserver_port := 11002;
LOCALCONST numupdate_rate := 0.10; !broadcast rate(sec)

LOCALVAR socketdevserver_socket;
LOCALVAR socketdevclient_socket;

PROC main()

  TPWrite "StateServer: Waiting for connection.";
  ROS_init_socketserver_socket, server_port;
  ROS_wait_for_clientserver_socket, client_socket;

  WHILE (TRUE) DO
    send_joints;
    WaitTime update_rate;
  ENDWHILE

```

Fig. 4.15. ROS\_stateServer module.

## 4.4 Gazebo

One of the many advantages of ROS as a robotic framework is that several tools for robot testing can be added to it. Gazebo, a simulation robotic environment, is one of those tools. It was designed as a standalone graphical application that does not require ROS to operate, and can be used with other robotics development platforms, but, because of the widespread adoption of ROS, a special purpose Gazebo package has been created for it.

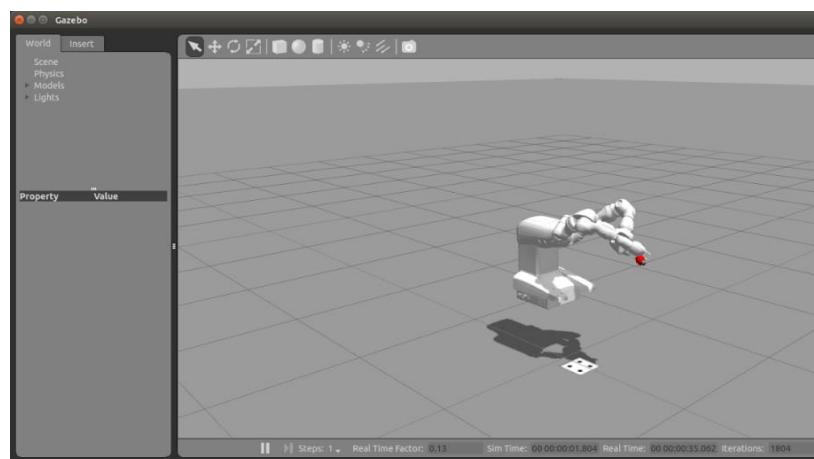


Fig. 4.16. Gazebo environment.

Gazebo brings together a set of functionalities perfect for testing robotics controls, algorithms, routines and behavior. It includes physics engines which makes it able to simulate body mechanics and dynamics. A number of pluggings have been developed to widen the options, for example, small pieces of code, wrapped as ROS packages that simulate the behavior of real sensors, can be include in the spawned virtual world just by adding the filename element of the plugging. A general overview of the interconnection of the different elements for the testing of the project algorithms in Gazebo is shown in figure 4.17.

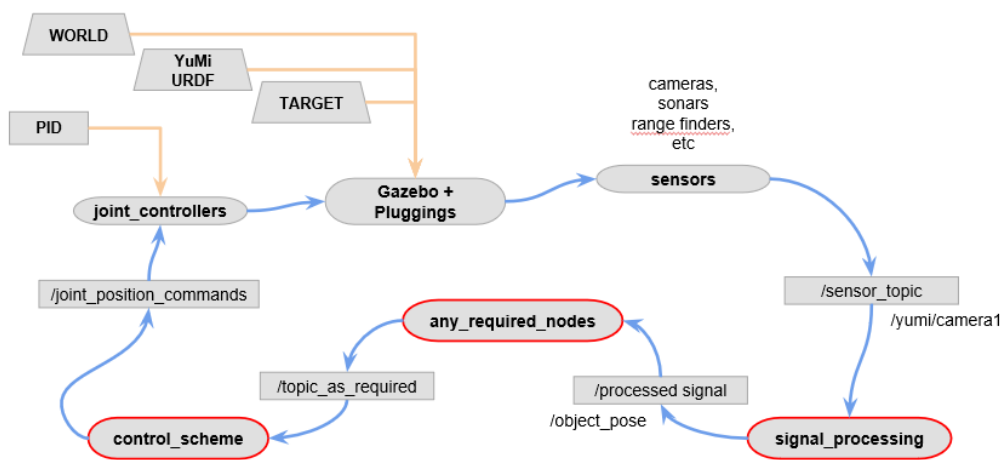


Fig. 4.17. Gazebo – ROS framework overview.

In the diagram, **Gazebo + Plugins**, which is the central node, contains the plant, the environment on which this plant will act and any other required elements, all those pieces: **WORLD**, **YuMi** and **TARGET**, are included in gazebo via definition of several parameters, as will be explained after. The plant receives commands from an existing low level control node called **joint\_controllers**, which is an implementation of a PID controller whose parameters can be tuned with a simple XML file. The plant also sends signal through existing **sensors**. The lower part of the loop is the implementation of the higher level control, which is the object of the project. The diagram shows some of the topics and nodes created for that aim.

The way of including all the different elements that will exist in that simulated world is another

good feature of Gazebo. Robots, sensors, and in general, any physical body that has to be spawned, is specified by a simple defined structure, either in a SDF (Simulation Description Format) or URDF (Universal Robot Description Format) XML file.

```
<link name="link_1.l">
  <inertial>
    <origin xyz="0 -0.03 0.12" rpy="0 0 0"/>
    <mass value="0.1"/>
    <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0" izz="0.01" />
  </inertial>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://yumi_description/meshes/link_1.stl"/>
    </geometry>
    <material name="Grey"/>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh
        filename="package://yumi_description/meshes/coarse/link_1.stl"/>
      </geometry>
      <material name="Grey"/>
    </collision>
  </link>

  <joint name="joint_1.l" type="continuous">
    <parent link="body"/>
    <child link="link_1.l"/>
    <origin xyz="0.05355 0.0725 0.41492"
      rpy="0.9781 -0.5716 2.3180"/>
    <axis xyz="0 0 1"/>
    <dynamics damping="3.0"/>
  </joint>
```

**Fig. 4.18.** Description of one of YuMi's links and joints.

All the links and joints of YuMi are described in its URDF, specifying the physical features of the links, with three main attributes: visual, collision and inertial, they assign respectively, the graphical appearance to render the link, the structure used to calculate the collision frontiers

of the link and its dynamics characteristic of distributed weight (Figure 4.18).

When the physical characteristics of the link and joint have been defined, in order to perform the simulation in Gazebo, then the mechanical and dynamical features of each joint, and the desired controller to be used to move it, just as in a real robot, should be specified (Figure 4.19).

Likewise, each sensor has a specific structure and attributes to be defined in the URDF. In the case of the project, a camera will be spawned in the virtual world to estimate the position of YuMi's TCP and control the movement of the arm.

```
<transmission name="tran_1.l">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="joint_1.l">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor_1.l">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>

<gazebo reference="link_1.l">
  <material>Gazebo/White</material>
  <selfCollide>true</selfCollide>
</gazebo>

<gazebo>
  <plugin name="gazebo_ros_control"
    filename="libgazebo_ros_control.so">
    <robotNamespace>yumi</robotNamespace>
    <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
  </plugin>
</gazebo>
```

**Fig. 4.19.** Description of the joint mechanics and controller.

Important parameters of the camera are set in its section of the XML file (Figure 4.20). The

update rate sets the number of frames per second that will be acquired by the sensor. Horizontal field of view (horizontal\_fov) is the parameter that, through equation (4.1) defines the focal distance in the camera.

```
<gazebo reference="camera_head">
  <sensor type="camera" name="camera1">
    <update_rate>30.0</update_rate>
    <camera name="head">
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>800</width>
        <height>800</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
    </camera>

    <plugin name="camera_controller"
      filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <cameraName>yumi/camera1</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info
      </cameraInfoTopicName>
      <frameName>camera_head</frameName>
      <hackBaseline>0.07</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
    </plugin>
  </sensor>
</gazebo>
```

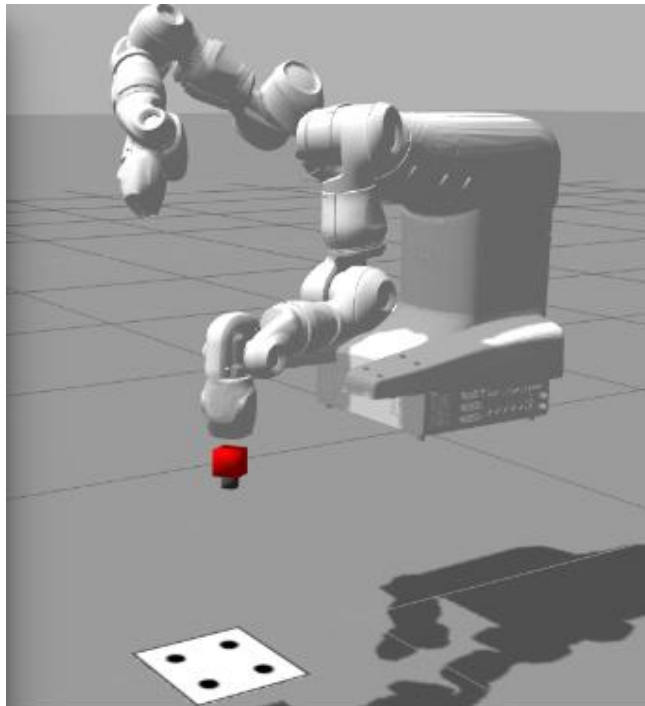
**Fig. 4.20.** Description of the camera sensor.

The image element of the description includes the size of the matrix and its format (either a

color or grayscale image). This plugging allows even the inclusion of some external disturbances as noise, by setting its type and its parameters accordingly, in the case of the camera shown, Gaussian noise distribution with 0 mean and 0.007 of standard deviation. In the case of cameras that have lenses, parameters for setting the several types of distortion that they produce in the image can also be set.

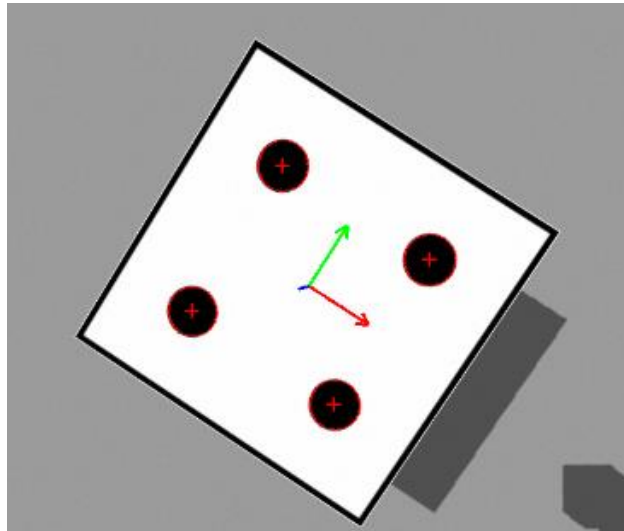
$$\text{horizontal field of view} = 2 * \tan^{-1}\left(\frac{0.5 * \text{width}}{\text{focal distance}}\right) \quad (4.1)$$

Once all the features of the objects that will be spawn in the virtual world have been written in the XML file, Gazebo can be started and the simulated robot will be ready to receive commands from ROS nodes with the designed algorithms and control schemes.



**Fig. 4.21.** YuMi spawn in Gazebo.

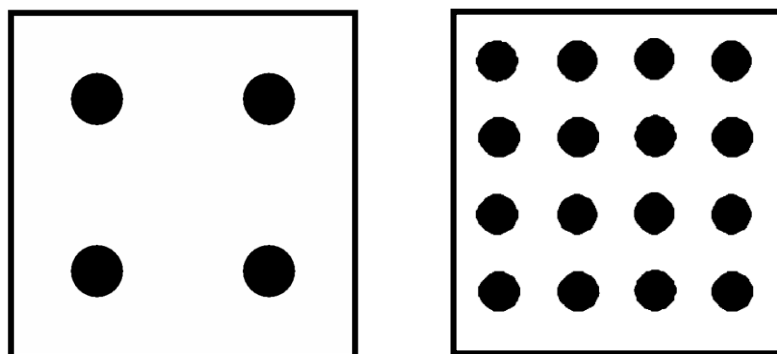
Simulated YuMi looks just as the real mechanism with a red camera in its left arm (Figure 4.21). The image being acquired by the camera in that robot pose is shown in figure 4.22.



**Fig. 4.22.** Image transmitted by the simulated camera

## 4.5 Target Patterns

Two graphical patterns have been used as target objects of the control algorithm, and thus, two approaches for the computer vision section of the system. A four blob symmetrical pattern was defined as the target for the simulation; in this case, as will be detailed later, the user selects the order of each blob, once this is done, the algorithm will track the blobs, the selected order specifies the pose of the object frame inside the pattern. For the real robot implementation, is used a symmetrical circles pattern, the algorithm detects it and assign the frame pose automatically.



**Fig. 4.22.** Four blobs pattern at left and symmetrical circles at right.

## 4.6 ROS Nodes

The ROS nodes, that compose the control loop of the system, have been implemented in the C++ programming language, they can be grouped in two categories, both have common image processing and control algorithm, but have differences in the definition of its publishers and subscribers. The nodes that interact with the real robot in the laboratory subscribe and publish one kind of topics; they receive messages coming from a real camera and will send motion commands to the real mechanism. On the other hand, the nodes for the simulated environment will get their messages from the sensor plugging and send the commands to the control plugging in Gazebo, so it is clear that the two groups of topics are different.

Then, within those categories, the control loop is implemented through two specific nodes. One that can be labeled as the sensor and signal processing node and other that implements the controller. The first type is in charge of acquiring the images with the camera, process that image, calculate the pose of the camera with respect to the target and publish that pose as a vector, which contains the position and orientation information, in the “ROS network”. The second type, the controller, subscribes to that pose vector, calculates the motion commands based on the specific used algorithm and publish that command to the network (Figure 4.23).



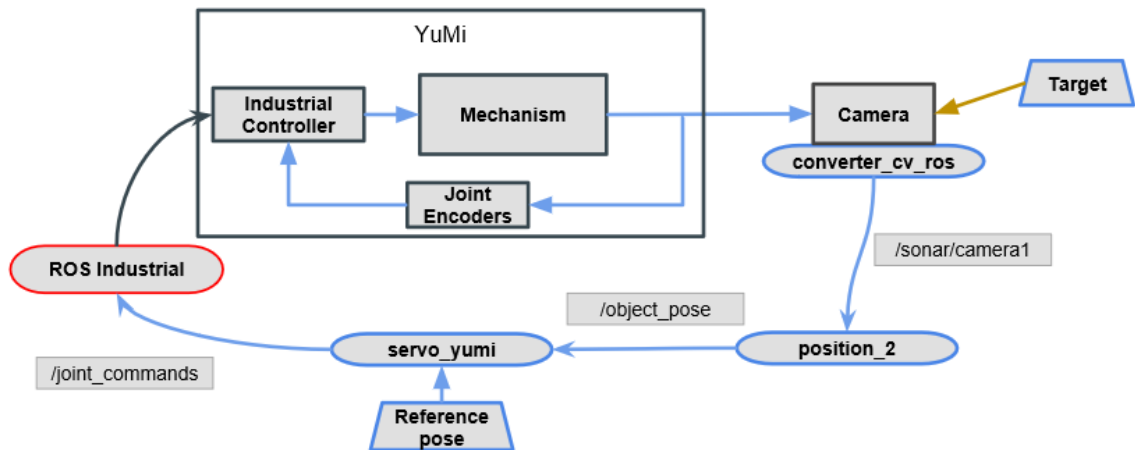


Fig. 4.23. Structure of the loop for controlling the real robot, it is composed of ROS nodes.

#### 4.6.1 Human joint limits and robot joint limits mapping

In order to bring the behavior of the robot further closer to a human coworker, restrictions in its joint movements have been included in the code. These correspond to a combination of the physical limits in joint motion and ranges of motion that are considered comfortable for a human worker executing manual labor.

For the human shoulder adduction (towards the body) and abduction (away from the body) movements, it has been found a range with the limits  $[-45^\circ, 130^\circ]$ , with the  $0^\circ$  point located in a vertical plane along the side of the trunk. The comfortable range is  $[90^\circ, 130^\circ]$ . The point of the zero for the human joints is not the same as for the robot. Then, by using RobotStudio, an empirical mapping can be done between human and robot joints

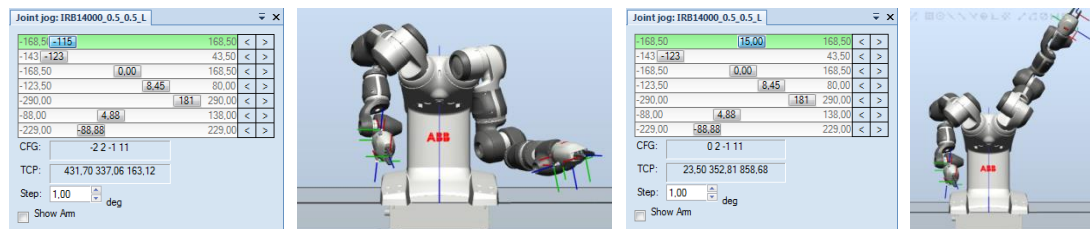
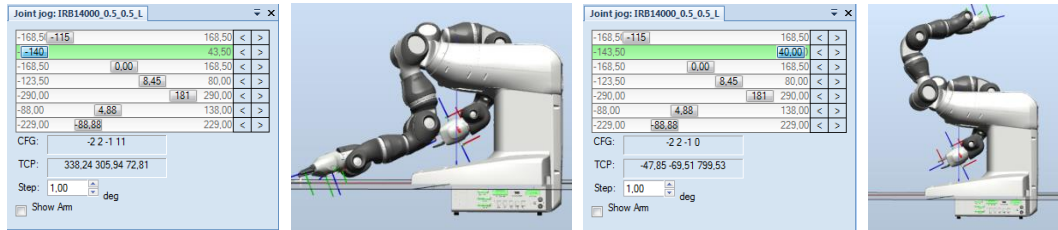


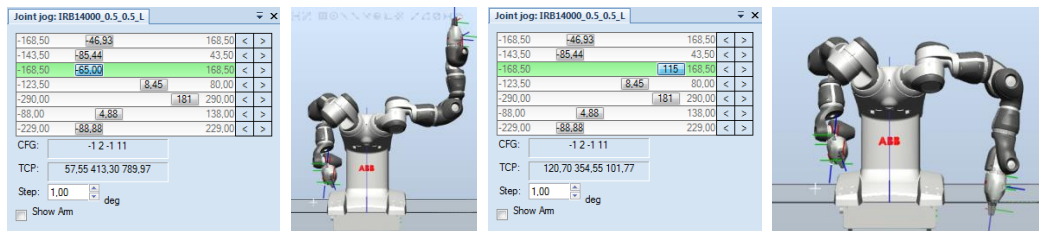
Fig. 4.24. Shoulder adduction-abduction.

The shoulder extension-flexion maximum human range is  $[-90^\circ, 135^\circ]$ , the comfortable range is  $[90^\circ, 135^\circ]$ , with  $0^\circ$  along the side of the trunk.



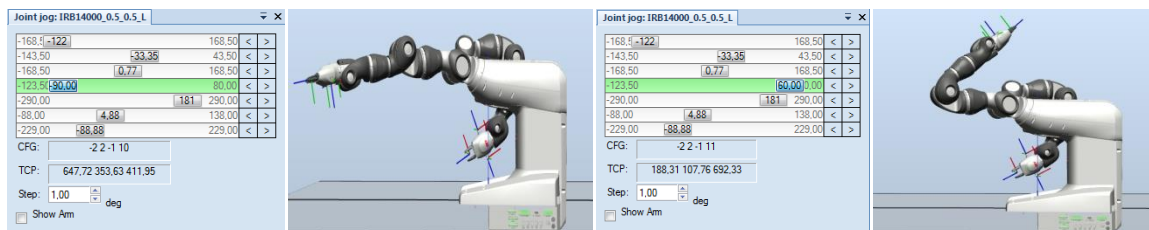
**Fig. 4.25.** Shoulder extension-flexion.

Human shoulder rotation movement goes in the maximum range of  $[-90^\circ, 150^\circ]$ , and comfortable range of  $[0^\circ, 90^\circ]$ .



**Fig. 4.26.** Shoulder rotation.

Finally, elbow extension-flexion is in the maximum range of  $[0^\circ, 150^\circ]$  and comfortable range of  $[0^\circ, 90^\circ]$ , the zero  $0^\circ$  position is the elbow fully extended.



**Fig. 4.27.** Elbow extension-flexion.

## 4.6.2 vp\_simulation

This node implements the sensor and signal processing phase within the control loop of the

simulation environment. It is programmed to detect and assign a user defined coordinate system to the target object, specifically for the four blobs pattern. The construction of the node is based on four main blocks, first, the section for creating the ROS node, as well as the publisher and subscriber contained in it. In this case, the node will subscribe to the topic **/yumi/camera1/image\_raw**, which is the one being published by the simulated camera created with the ROS plugin. The instantiated publisher, will send a topic called **/object\_pose**, a ROS multi-array message, composed of a twelve element vector, that is, three elements for specifying the position of the target (the four blob pattern), nine elements for the matrix that defines rotation and a Boolean element indicating whether the pattern has been found and its pose calculated (= 1) or not (= 0).

The code makes use of objects instantiated from classes of two C++ libraries: OpenCV and ViSP (Visual Servoing Platform). These objects have functions that permit the manipulation of images as matrixes formed of floating point numbers. Each library implements the matrixes in its own format; they have specific headers, pointers and attributes, and are not interchangeable, specific functions must be implemented to pass from one format to the other..

Then, the second component of the node, is a function that transform the ROS image format received by the subscriber, and performs two successive transformations, first, it translates this message into an OpenCV image format, and then converts it into ViSP format, this is done to take advantage of the classes and functions of this last library for detecting and assigning frames in the used pattern.

The final component is a block that allows the assignation, via mouse clicking, of every blob of the image returned by the previously explained function, and then calls another function that takes as inputs the set coordinates of each blob with respect to the desired origin of the target frame and the user clicked order assigned for each blobs, delivering the pose of the

object frame. The message to be sent by the publisher is then built from this function output.

#### 4.6.3 converter\_cv\_ros

This node is in charge of acquiring the images from the CANON camera and publishing those images as a ROS message, it runs in the CAM PC, the topic of the message is called **/sonar/camera1**.

The CAM PC (labeled as *sonar* in the laboratory network) has a hub card where several cameras are connected; the code written in the node, instantiate a *VideoCapture* class of OpenCV indicating the device that must be opened depending on the number of port of the camera that is going to be used. Then it uses functions of the ROS package *cv\_bridge*, these functions convert the matrix image structure of OpenCV into a ROS message. This node serves only as the connection and transmission interface of the camera with the ROS network, the image acquired is sent through the network without any further processing.

#### 4.6.4 position\_2

The node **position\_2** is the implementation of the sensor in the control loop of the real robot. Unlike **vp\_simulation**, which takes a topic published from the Gazebo plugin, this node is subscribed to the topic **/sonar/camera1** whose messages come from the camera converter, and contain real images.

The input of this node is an image that is processed by computer vision functions, and then delivers, as output, the pose of the target object frame with respect to the camera, this pose is published in a topic called **/object\_pose**. The message composed of a 1x16 array that contains: the nine elements of the rotation matrix, the three elements of the translation vector, a Boolean variable that signals whether the pattern has been found or not, and three final elements that are the angles of rotation of the around each of the axes of the camera

frame, this angles come from the rotation matrix, and are published mainly for plotting purposes.

The arrangement of the code is similar to the one used in the node **vp\_simulation**, a section where the ROS node is created, with the required publishers and subscribers. A second section is formed by a function that is called every time a new message arrives and converts back the ROS message into an OpenCV image structure. The third section has the function that detects the pattern and calculates the pose of the object frame.

#### 4.6.5 **servo\_simulation and servo\_yumi**

The high level control of the real robot is implemented in the node **servo\_yumi** and in node **servo\_simulation** for the virtual one. As usual, within the `main()` function, the node is created with its publishers and subscribers.

In the real framework the node subscribes to two topics: **/object\_pose** coming from the vision sensor (the **position\_2** node) and **/left\_arm/joint\_states**, this topic carries messages coming from the robot through the ROS\_industrial framework and brings information of the position of each joint; **servo\_yumi** publishes the commands for the motion of all the joints in a single topic called **/left\_arm/joint\_path\_command**, the ROS type of messages transported in this topic is a structure called **trajectory\_msgs::JointTrajectory**, it contains an array of seven elements, one per joint.

For the simulation case, the control sends the commands in one topic for every joint, it is done in this manner because of the definitions done in the *URDF* and *launch* file of the virtual robot, each joint controller expects a specific topic. Then for example, the command for the first joint in the left arm of the manipulator is in the topic called **/yumi/joint\_1\_position\_controller/command**, the second joint topic is called **/yumi/joint\_2\_position\_controller/command**, and so on for the other joints. Every topic

contains a **std\_msgs::Float64** ROS type message; it is one double precision floating point number.

Then a section comes where the kinematic chain of the robot is created. This is done through instantiation of KDL classes whose functions extract the information from the ROS *Parameter Server*, the kinematic chain is the input to functions for getting forward, inverse kinematics, and Jacobian solvers.

Next, the ROS loop is started; the rate of this loop defines the frequency of subscription and publication of messages of the node. It depends on the frequency at which the low level controller is capable of receiving joint commands. The closer is this rate to the rate at which the images are acquired and the **/object\_pose** message is being published, the better the performance of the control, otherwise there will be a bigger latency between the signal sent by the sensor (**vp\_simulation** or **position\_2**) and the processing done by the control.

In the case of **servo\_simulation**, the loop rate has been defined equal to the rate of the ROS loop in **vp\_simulation**, so the control and the sensor can be considered to be synchronized. That is not possible for **servo\_yumi**, due to speed constraints in: the socket connection, the ability of RAPID to process incoming messages and an issue with the execution of the program in the industrial controller (this will be commented in a later section). So the rate of the control block in the real robot is much lower than the publishing rate of its sensor, **position\_2**.

With the information of instantaneous rotation and translation of the object; a control law based on (3.15) is implemented, thus calculating the needed Cartesian velocities of the TCP for positioning the target in the goal pose. There are restrictions to this velocities, if the robot is in a singular configuration, exist directions of the velocity vector that are impossible to fulfill by the robot. A function has been implemented that, if the robot is in a singular configuration,

calculates that “forbidden” velocity and compares it with the commanded velocity of the control algorithm, if the commanded velocity is close enough (with a defined tolerance) to the “forbidden” velocity, a new command velocity is calculated, which will be orthogonal to the originally obtained by the control law.

With the required Cartesian velocity obtained (either the original coming from the equation or after correcting for non possible velocities), the joint velocities are calculated through an implementation of a function for inverse Jacobian. In both, the simulation robot and the real one the low level controller expects joint position commands, so the obtained joint velocity command is converted to joint position by assuming a constant sampling time  $dt$ , that is:

$$\mathbf{q}_{commanded}(t) = \mathbf{q}_{commanded}(t - 1) + \dot{\mathbf{q}}_{obtained}(t) * dt \quad (4.2)$$

Especially in the case of the real robot, the definition of this  $dt$  largely affects the time evolution of the measured variables (the behavior will be seen in the next section).

Once the joints position command is obtained, another function takes the first four elements of the array and compare them with the limits for comfortable ranges of motion described in the previous section. The calculated joint position will be sent to the robot only if it is inside those limits.

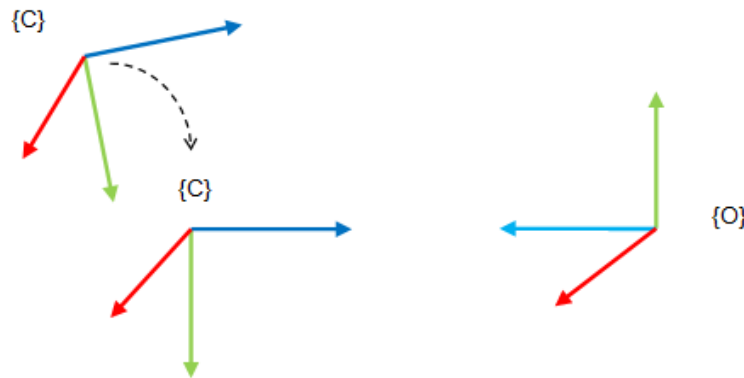




## RESULTS

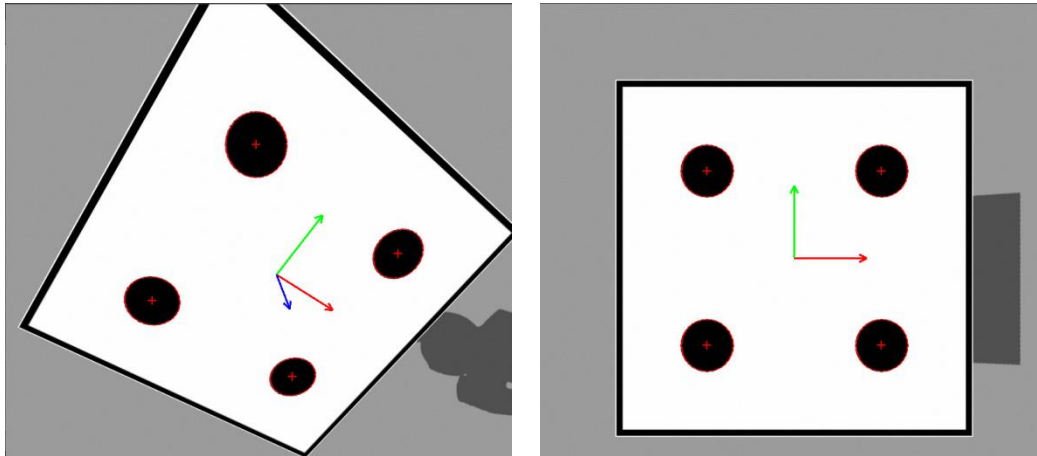
### 5.1 Simulation in Gazebo

Once all the nodes have been created and the parameters set, the simulation in Gazebo can be run. The objective, programmed in the code, is to position the camera frame  $\{C\}$  and the object frame  $\{O\}$  so there is a translation relation vector of  $[0, 0, 0.35]$ , that is, the origin of the object frame is to be in the origin of the X-Y plane and at 35 centimeters displaced in the Z direction of the camera frame. With respect to rotation, the aim is that the object frame be rotated a total of  $[-3.14, 0, 0]$  radians (that is  $[-180, 0, 0]$  degrees) in every axis of the camera frame.



**Fig. 5.1.** Goal pose of the camera frame with respect to the object frame.

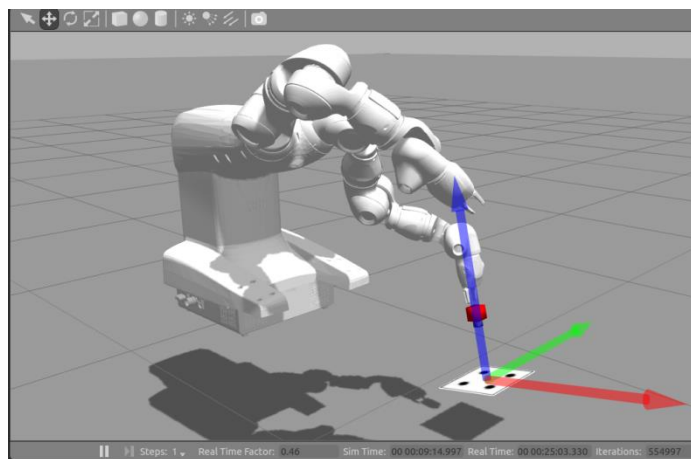
The figure 5.2 shows the images of the target pattern captured by the virtual camera hold by the robot with its left gripper. At the left, the starting pose, an instant before the algorithm starts sending motion commands to the joints. At the right, the result, once the servo control has been able to track the objective pose of the camera with respect to the object frame. It is seen that the control is able to exactly reach the defined objective pose.



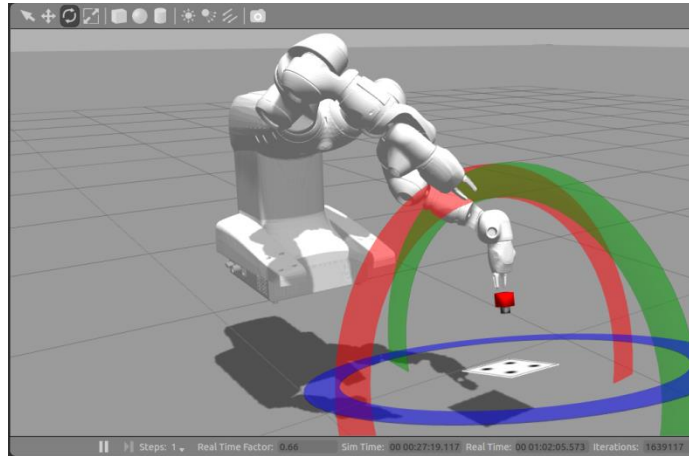
**Fig. 5.2.** Images of initial and final pose of the object frame.

The shown images are for a run of the algorithm in which the target object has not been moved.

However, the object can be moved “on-line” in the virtual environment to test the capacity of the algorithm to track that movement, both in position and orientation. The Gazebo interface allow these variations by selecting in the upper menu whether the body is to be translated or rotated, and then just dragging it along the depicted arrows (Figure 5.3 and 5.4)



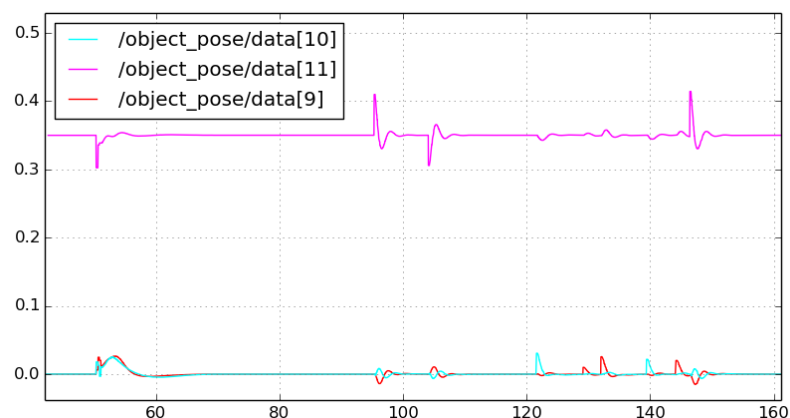
**Fig. 5.3.** Target position changed during execution.



**Fig. 5.4.** Target orientation changed during execution.

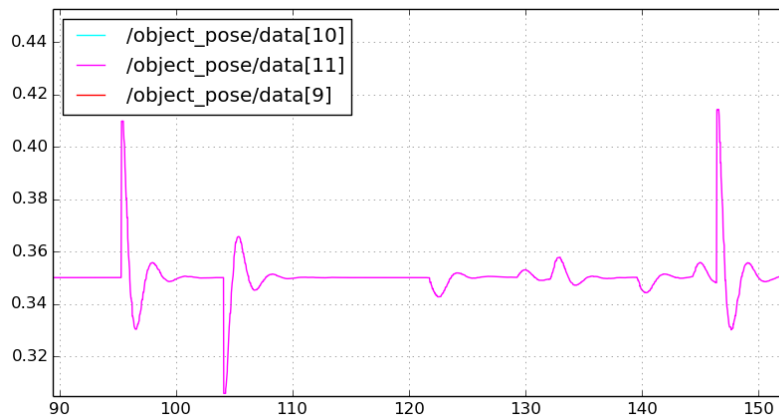
Effectively the control tracks the new positions of the object and moves the camera accordingly so it is positioned at the distance defined in the code. The evolution of the changes in position and the tracking behavior is shown, for the full run time in figure 5.5. The labels of each line correspond to the topic name and the structure message being published by the visual sensor node (**vp\_simulation**).

It is clear that when there is a perturbation in the system (that is, a forced change in the position of the object), the visual servo control regulates the position of the camera to converge to the desired values.



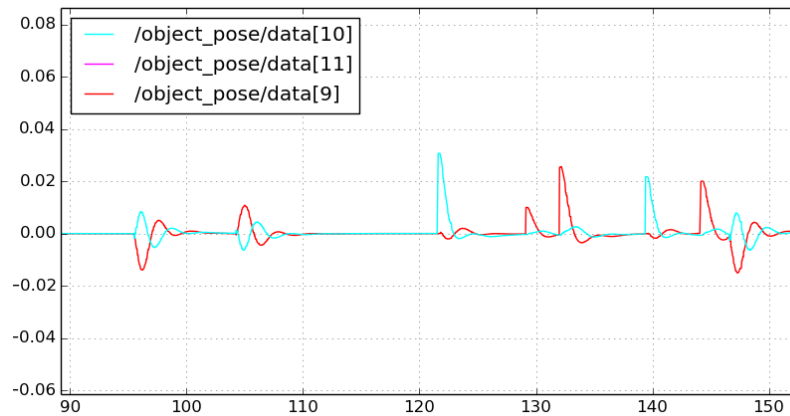
**Fig. 5.5.** Evolution of the position.

Zooming in a region of the plot (Figures 5.6 and 5.7), some details can be analyzed. Even though the position converges, there are some overshooting in the response. This may be due to several reasons. One of them is the fact that, the low level joint controllers (a PID for every joint) may need finer tuning, that is, it is probable that some of those oscillations come from the process of the PID trying to converge at every instant to the command joint position sent by the high level control. An empirical procedure has been followed to set the PID parameters at acceptable performance values since this fine tuning procedure is not a central part of the project, but the time behavior of the variables suggest that there is still room for improvement in the tuning of the PID.



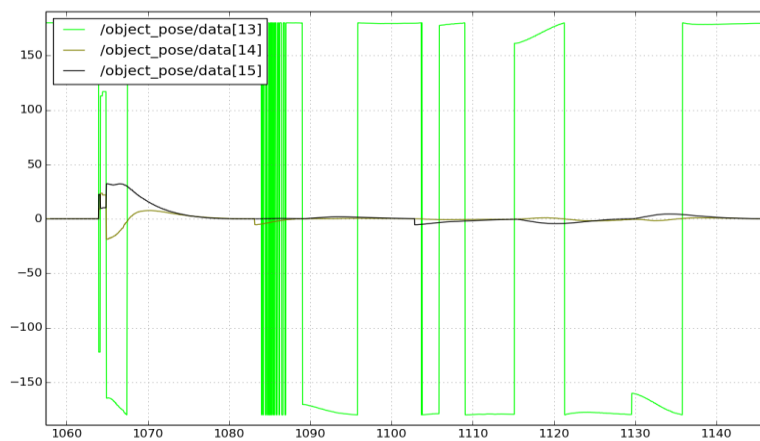
**Fig. 5.6.** Zoom to Z distance.

The range of disturbance that can be applied to the simulated system depends largely of the limited update capacity of Gazebo to produce a continuous stream of images when a body is being moved. If the movement is done abruptly, Gazebo losses the image for an instant, and then the tracking algorithm in the **vp\_simulation** node, loses the center points of the circles. This range can be appreciated in the maximum peak values applied in the X and Y direction, a deviation of no more than 4 centimeters have been applied to avoid that effect of losing the image during the tracking.



**Fig. 5.7.** Zoom to X and Y distances.

The results can be compared with the ones found in the literature; in [9] for example, an IBVS scheme was applied to an industrial application with very similar behavior; in [8] a PBVS was implemented in a 6 DOF manipulator in a laboratory setting. In both cases however, no tracking was done, the target object remain still during the whole test and the plotting finishes once the position has converged for the first time

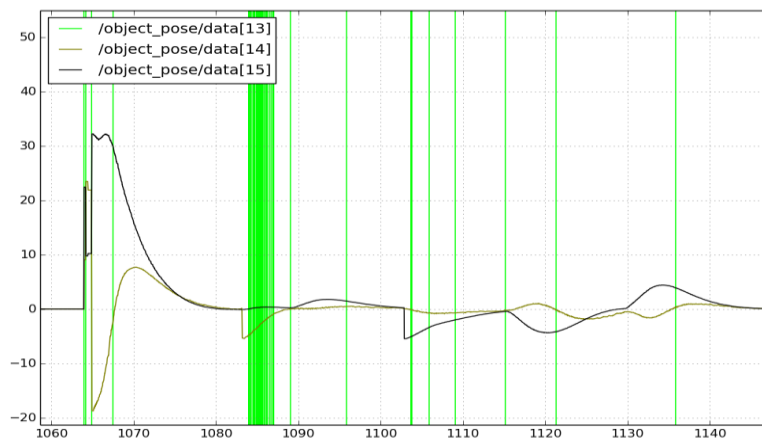


**Fig. 5.8.** Evolution of the orientation.

In a new run of the simulation, the behavior of the orientation is plotted and is depicted in figures 5.7 and 5.8. The set point for the angles was defined so that only a rotation in X axis was required to relate the camera frame with the object frame. This configuration allows

positioning the Z axis of each frame “face to face”.

The rotation in the Y and Z axes has good convergence toward the desired orientation. The algorithm manages to regulate them when faced to disturbances in both sides of the set point. On the other hand, the behavior of the rotation in X axis does not appear to be good. It seems to have large oscillations trying to converge to the points 180 and -180 degrees. This is due to the fact that the visual sensor takes the value of rotation of the frame positive or negative for small variations around the set point. For example if the pose is at -178 degrees, the camera is heading toward the -180 degrees point and passes two degrees after the point and the sensor will mark 178 degrees. This was accounted by in the servo algorithm, so the set point actually changes to -180 to 180 as required.



**Fig. 5.9.** Zoom to Y and Z rotation angles.

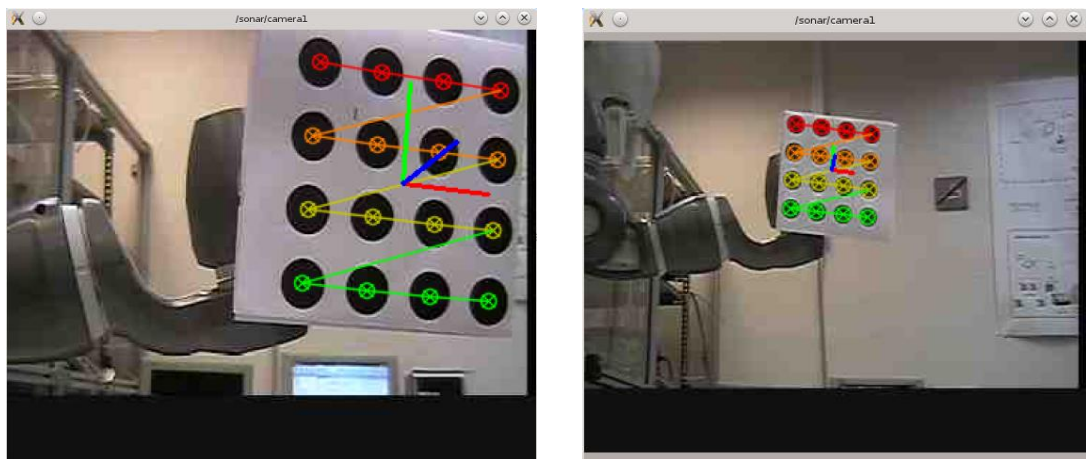
## 5.2 Real robot

After successfully testing the code in the simulation environment, it was implemented in the real robot. In this case, the target object is the grid of circles pattern and the configuration used is the eye-to-hand.



**Fig. 5.10.** Goal pose of the object frame with respect to the camera frame.

Figure 5.11 presents examples of the image acquired by the CANON camera when the object is located at different distances from it. It can be seen that the computer vision algorithms detects the pattern and assigns the required frame correctly. It is worth mentioning the rather narrow field of view of the available camera; that characteristic reduces the amplitude of the useful space for the test.



**Fig. 5.11.** Images of initial positions of the pattern, located close and far from the camera

The target object is located at an initial distance and orientation with respect to the camera (Figure 5.12, left). Once the ROS network is initialized and the socket communication between the industrial controller and ROS PC established, the algorithm starts sending

commands to the real robot, moving the gripper, that holds the pattern, towards the defined set point, the final result is shown in figure 5.12, right.



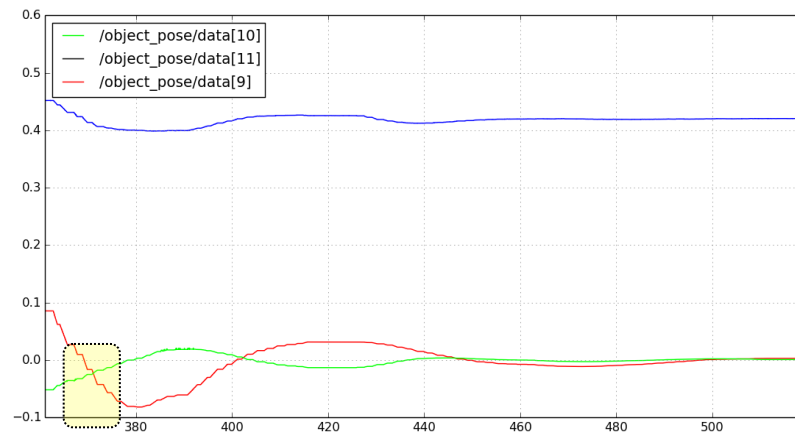
**Fig. 5.12.** The initial and final position of the pattern.

The time evolution of the target object coordinates with respect to the camera, is plotted in figure 5.13. The set point vector was specified as  $[0, 0, 0.42]$ , that is, in the origin of the X, Y plane and at 42 centimeters in the Z direction. It is seen that the position converges to the set point.

The movements in the Z and Y direction present small overshooting and softer approach towards the desired position. On the other hand, the movement in the X axis has a larger overshoot and a greater settling time.

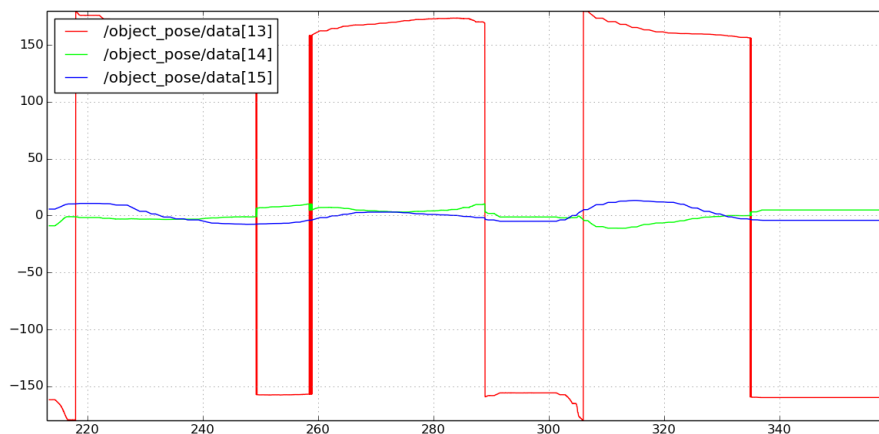
In general, the movement of the robot is not smooth; the positions move forward in small steps, that behavior is highlighted in the plot (yellow rectangle). This feature was not present in the robot simulated in Gazebo, nor when sending motion commands to the virtual station created in RobotStudio. The origin of those small steps is due to the fact that every time a new joint command is sent to the real robot, execution of the RAPID program, loaded in the industrial controller, stops and it must be reset again; after resetting the program execution, the robot continues with the latest sent joint command.



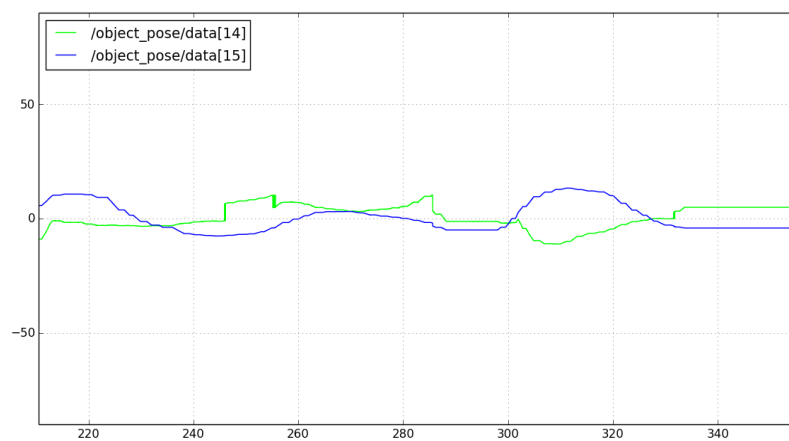


**Fig. 5.13.** Evolution of the position.

When seeing the behavior of the frame angles (Figure 5.14 and 5.15). The issue commented before becomes more evident. Even though the control directs every controlled variable towards its set point, the time gap between the command generated by the algorithm and the one executed by the industrial controller prevents the angles to settle completely. This is more of an implementation problem, and not a problem of the controller itself; the algorithm, despite those synchronization issues, maintain the angles within a range and seeks to position them in the desired value.



**Fig. 5.14.** Evolution of the orientation in the three axes.



**Fig. 5.15.** Evolution of the orientation in axes Y and Z.

## ENVIRONMENTAL IMPACT

This project has been carried out in the Robotics Laboratory of IOC, in the ETSEIB, floor 11. It has dealt with the development of control algorithms and its run in computer platforms. No significant impact has been done to the environment other than the one derived from the power consumption of desktop computers and an industrial small size robot. This can be summarized in the following table:

<i><b>Equipment</b></i>	<i><b>Power [kW]</b></i>	<i><b>Units</b></i>	<i><b>Time [h]</b></i>	<i><b>Energy [kWh]</b></i>
Computer ROS PC	0.360	1	1008	362.88
Computer WIN PC	0.360	1	302	108.72
Computer CAM PC	0.360	1	201	72.36
Robot	1.500	1	302	453
<b>TOTAL</b>				<b>996.96</b>

There have not been any sizable residuals due to the execution of the project; a small amount of paper has been recycled after use.



## COST ANALYSIS

The only cost derived from the project may refer to the depreciation of the laboratory equipment due to its use. No consumables were used. Assuming a lifespan of 10 years, then 11 months per year, 24 days per month and 10 hours per day of normal use of the equipment, estimated time depreciation is shown in the following table:

<i><b>Equipment</b></i>	<i><b>Cost [€]</b></i>	<i><b>Units</b></i>	<i><b>Time [h]</b></i>	<i><b>Percentage of total time</b></i>	<i><b>Total Cost [€]</b></i>
Computer ROS PC	1000	1	1008	0.038	38.00
Computer WIN PC	1000	1	302	0.011	11.00
Computer CAM PC	1000	1	201	0.007	7.00
Robot	30000	1	302	0.011	330.00
<b>TOTAL</b>					<b>386.00</b>



## CONCLUSIONS AND FUTURE WORK

This project has implemented a visual based servo control loop in an industrial collaborative robot. Before running the algorithms in the real framework, they have been tested in a simulation environment specifically created for that end.

Several robotics, computer vision and programming tools have been used in order to achieve the objectives of this project. Two sets of these tools can be separated. Since the used manipulator (YuMi) is an industrial robot of recent introduction to the market, so it is still mostly a closed platform, it has been necessary to attain familiarity and knowledge with the proprietary programming language and environment of the robot, which are Rapid and RobotStudio. Those represent one of the types of used tools, the other group is formed by the open source languages, programs and framework: C++ (with libraries OpenCV, KDL), ROS, Gazebo. Through the process of making the project it has been possible, using both newly created and existing codes and routines, to merge those two a priori non compatible set of tools. This merger is a first step in further developments using YuMi as a research platform, which will be also facilitated by future additions, from the manufacturer, of yet more communication and external control options. One of them is the External Guided Motion module recently developed for YuMi by ABB, this addition will allow a more straightforward, fast and direct way of sending motion commands to the robot.

The use of visual servoing has been proven, by the project, a valuable option to give a cobot the capacity to react to changes in its work environment. With this control scheme the robot has more flexibility in the case of pieces or tools changing position during the execution of its working cycle. Also, in an usual vision guided control there is the necessity of having a constant and known pose transformation between the camera and the robot frame, if this

relation is not perfectly calibrated (spatial calibration) or if the camera moves from its expected position, the control will fail in reaching the objective point, but in a eye-in-hand strategy, the robot holds the camera, so it knows at every instant the transformation relating the camera frame. On the other hand, the requirements for precise visual calibration and the capacity to extract an object position from the image are stricter in the case of visual servoing since the camera is the main sensor of the robot. As the cameras and computer vision algorithms improve in precision and speed, these requirements will be less an impediment for a widespread use of visual servoing.



## BIBLIOGRAPHY

- [1] Chaumette, F., Hutchinson, S., *Visual servo control, Part I: Basic approaches*, IEEE Robotics and Automation Magazine, 13(4), pp.82-90, 2006.
- [2] Chaumette, F., Hutchinson, S., *Visual servo control, Part II: Advanced approaches*, IEEE Robotics and Automation Magazine, 14(1), pp.109-118, 2007.
- [3] Corke, P., *Robotics, Vision and Control: Fundamental Algorithms in Matlab*, Springer-Verlag, 2011.
- [4] Corke, P., *Visual Control of Robots: High Performance Visual Servoing*, Wiley, N.Y., 1996.
- [5] Craig, J.J., *Introduction to Robotics: Mechanics and Control*, 2013, 3<sup>rd</sup> Edition.
- [6] Forsyth, D., Ponce, J., *Computer Vision, a Modern Approach*, Pearson Education, 2012 2nd Edition.
- [7] Hutchinson, S., Hager, G., Corke, P., *A tutorial on visual servoing*, IEEE Transactions on Robotics and Automation, 12(5), pp.651-670, 1996.
- [8] Janabi-Sharifi, F., *Visual Servoing: Theory and Applications*. In: Cho, H.S. ed., *Opto-Mechatronics Systems-Handbook*, 1<sup>st</sup> ed. CRC Press, 2003.
- [9] Kosmopoulos, D. I., *Robust Jacobian matrix estimation for image-based visual servoing*, Robotics and Computer-Integrated Manufacturing, vol.27, pp.82-87, 2011.
- [10] Malis, E., Chaumette, F., Boudet, S., *2-1/2D visual servoing*, IEEE Transactions on Robotics and Automation, vol.15, pp.238-250, 1999.
- [11] OpenCV.org, <http://docs.opencv.org/2.4/index.html>, 2016.
- [12] Orocos.org, [http://docs.ros.org/indigo/api/orocos\\_kdl/html/](http://docs.ros.org/indigo/api/orocos_kdl/html/), 2016.
- [13] The ABB Corporation, *Operating manual – RobotStudio*, Document ID: 3HAC032104-001, Revision: Q, 2015.
- [14] The ABB Corporation, *Operating manual – IRC5 with FlexPendant*, Document ID: 3HAC050941-001, Revision: A, 2015
- [15] The ABB Corporation, *Product manual – IRB 14000-0.5/0.5*, Document ID:

3HAC052983-001, Revision: B, 2015.

[16] The ABB Corporation, *Technical reference manual – RAPID Instructions, Functions and Data types*, Document ID: 3HAC050917-001, Revision: B, 2015.

[17] visp.inria.fr, <http://visp-doc.inria.fr/doxygen/visp-2.9.0/modules.html>, 2016.